

c o n f e r e n c e

*proceedings*

## **FREENIX Track**

**1999 USENIX**

**Annual Technical Conference**

*Monterey, California, USA  
June 6–11, 1999*

Sponsored by

**USENIX**  
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

For additional copies of these proceedings contact:

USENIX Association  
2560 Ninth Street, Suite 215  
Berkeley, CA 94710 USA  
Phone: 510 528 8649  
FAX: 510 548 5738  
Email: [office@usenix.org](mailto:office@usenix.org)  
WWW URL: <http://www.usenix.org>

The price is \$22 for members and \$30 for nonmembers.  
Outside the U.S.A. and Canada, please add  
\$18 per copy for postage (via air printed matter).

© 1999 by The USENIX Association  
All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. USENIX acknowledges all trademarks herein.

ISBN 1-880446-32-4

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.

**USENIX Association**

**Proceedings of the  
FREENIX Track**

**1999 USENIX Annual Technical Conference**

**June 6–11, 1999  
Monterey, California, USA**

## **FREENIX Track Organizers**

### **FREENIX Track Committee**

**Chair:** Jordan Hubbard, *FreeBSD*

David Greenman, *FreeBSD*

John Ioannidis, *AT&T Labs—Research*

Angelos D. Keromytis, *OpenBSD*

Kirk McKusick, *Author & Consultant*

Jason Thorpe, *NetBSD*

Nathan Torkington, *Consultant*

Theodore Ts'o, *MIT*

### **The USENIX Association Staff**

# Contents

## **FREENIX Track:** **1999 USENIX Annual Technical Conference** **June 6–11, 1999** **Monterey, California, USA**

<b>Index of Authors</b> .....	vii
-------------------------------	-----

<b>Message from the Program Chair</b> .....	ix
---	----

### **Wednesday, June 9**

#### **File Systems**

*Session Chair: David Greenman, FreeBSD*

Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem .....	1
<i>Marshall Kirk McKusick, Author and Consultant; Gregory R. Ganger, Carnegie Mellon University</i>	

Design and Implementation of a Transaction-Based Filesystem on FreeBSD .....	19
<i>Jason Evans, The Hungry Programmers</i>	

The Global File System: A Shared Disk File System for *BSD and Linux .....	not available
<i>Kenneth Preslan and Matthew O'Keefe, University of Minnesota; John Lekashman, NASA Ames</i>	

#### **Device Drivers**

*Session Chair: David Greenman, FreeBSD*

Standalone Device Drivers in Linux .....	27
<i>Theodore Ts'o, MIT</i>	

Design and Implementation of Firewire Device Driver on FreeBSD .....	41
<i>Katsushi Kobayashi, Communication Research Laboratory</i>	

newconfig: A Dynamic-Configuration Framework for FreeBSD .....	53
<i>Atsushi Furuta, Software Research Associates, Inc.; Jun-ichiro Hagino, Research Laboratory, Internet Initiative Japan Inc.</i>	

#### **File Systems**

*Session Chair: Theodore Ts'o, M.I.T.*

The Vinum Volume Manager .....	57
<i>Greg Lehey, Nan Yang Computer Services Ltd.</i>	

Porting the Coda File System to Windows .....	69
<i>Peter J. Braam, Carnegie Mellon University; Michael J. Callahan, The Roda Group, Inc.; M. Satyanarayanan and Marc Schnieder, Carnegie Mellon University</i>	

A Network File System over HTTP: Remote Access and Modification of Files and files .....	75
<i>Oleg Kiselyov</i>	

## Thursday, June 10

### Security

*Session Chair: Angelos D. Keromytis, OpenBSD*

A Future-Adaptable Password Scheme .....81  
*Niels Provos and David Mazières, The OpenBSD Project*

Cryptography in OpenBSD: An Overview .....93  
*Theo de Raadt, Niklas Hallqvist, Artur Grabowski, Angelos D. Keromytis, and Niels Provos, The OpenBSD Project*

Minding Your Own Business: The Platform for Privacy Preferences Project and Privacy Minder .....103  
*Lorrie Faith Cranor, AT&T Labs—Research*

### Networking

*Session Chair: Nathan Torkington, Consultant*

Trapeze/IP: TCP/IP at Near-Gigabit Speeds .....109  
*Andrew Gallatin, Jeff Chase, and Ken Yocum, Duke University*

Managing Traffic with ALTQ .....121  
*Kenjiro Cho, Sony Computer Science Laboratories, Inc.*

Opening the Source Repository with Anonymous CVS .....129  
*Charles D. Cranor, AT&T Labs—Research; Theo de Raadt, The OpenBSD Project*

### Business

*Session Chair: John Ioannidis*

Open Software in a Commercial Operating System .....139  
*Wilfredo Sánchez, Apple Computer, Inc.*

Business Issues in Free Software Licensing .....143  
*Donald K. Rosenberg, Stromian Technologies*

Doing Well, Doing Good, and Staying Sane: A Hybrid Model for Sustainably Producing Innovative Open Software .....not available  
*Nathaniel S. Borenstein, Joseph Hardin, and Marshall Van Alstyne, School of Information, University of Michigan*

### Systems

*Session Chair: Kirk McKusick, Author and Consultant*

Sendmail Evolution: 8.10 and Beyond .....149  
*Gregory Neil Shapiro and Eric Allman, Sendmail, Inc.*

The GNOME Desktop Project .....not available  
*Miguel de Icaza, Universidad de Mexico*

Meta: A Freely Available Scalable MTA .....159  
*Assar Westerlund, Swedish Institute of Computer Science; Love Hörnquist-Åstrand, Dept. of Signals, Sensors, and Systems, KTH; Johan Danielsson, Center for Parallel Computers, KTH*

## Kernel

*Session Chair: Jordan Hubbard, FreeBSD*

Porting Kernel Code to Four BSDs and Linux .....	165
<i>Craig Metz, ITT Systems and Sciences Corporation</i>	
strncpy and strlcat—Consistent, Safe, String Copy and Concatenation .....	175
<i>Todd C. Miller, University of Colorado, Boulder; Theo de Raadt, The OpenBSD Project</i>	
pk: A POSIX Threads Kernel .....	179
<i>Frank W. Miller, Cornfed Systems, Inc.</i>	

## Friday, June 11

### Applications

*Session Chair: Jason Thorpe, NetBSD*

Berkeley DB .....	183
<i>Michael A. Olson, Keith Bostic, and Margo Seltzer, Sleepycat Software, Inc.</i>	
The FreeBSD Ports Collection .....	193
<i>Satoshi Asami, The FreeBSD Project</i>	
Multilingual vi Clones: Past, Now and the Future .....	195
<i>Jun-ichiro itojun Hagino, KAME Project; Yoshitaka Tokugawa, WIDE Project</i>	

### Kernel

*Session Chair: Jason Thorpe, NetBSD*

Improving Application Performance Through Swap Compression .....	207
<i>R. Cervera, T. Cortes, and Y. Bercerra, Universitat Politècnica de Catalunya—Barcelona</i>	
New Tricks for an Old Terminal Driver .....	219
<i>Eric Fischer, University of Chicago</i>	
The Design of the Dents DNS Server .....	223
<i>Todd Lewis, MindSpring Enterprises</i>	



## Index of Authors

Allman, Eric . . . . .	149	Keromytis, Angelos D. . . . .	93
Alstyne, Marshall Van . . . . .	not available	Kiselyov, Oleg . . . . .	75
Asami, Satoshi . . . . .	193	Kobayashi, Katsushi . . . . .	41
Åstrand, Love Hörnquist- . . . . .	159	Lehey, Greg . . . . .	57
Bercerra, Y. . . . .	207	Lekashman, John . . . . .	not available
Borenstein, Nathaniel S. . . . .	not available	Lewis, Todd . . . . .	222
Bostic, Keith . . . . .	183	Mazières, David . . . . .	81
Braam, Peter J. . . . .	69	McKusick, Marshall Kirk . . . . .	1
Callahan, Michael J. . . . .	69	Metz, Craig . . . . .	165
Cervera, R. . . . .	207	Miller, Frank W. . . . .	179
Chase, Jeff . . . . .	109	Miller, Todd C. . . . .	175
Cho, Kenjiro . . . . .	121	O'Keefe, Matthew . . . . .	not available
Cortes, T. . . . .	207	Olson, Michael A. . . . .	183
Cranor, Charles D. . . . .	129	Preslan, Kenneth . . . . .	not available
Cranor, Lorrie Faith . . . . .	103	Provos, Niels . . . . .	81, 93
Danielsson, Johan . . . . .	159	Raadt, Theo de . . . . .	93, 129, 175
Evans, Jason . . . . .	19	Rosenberg, Donald K. . . . .	143
Fischer, Eric . . . . .	219	Sánchez, Wilfredo . . . . .	139
Furuta, Atsushi . . . . .	53	Satyanarayanan, M. . . . .	69
Gallatin, Andrew . . . . .	109	Schnieder, Marc . . . . .	69
Ganger, Gregory R. . . . .	1	Seltzer, Margo . . . . .	183
Grabowski, Artur . . . . .	93	Shapiro, Gregory Neil . . . . .	149
Hagino, Jun-ichiro itojun . . . . .	53, 195	Tokugawa, Yoshitaka . . . . .	195
Hallqvist, Niklas . . . . .	93	Ts'o, Theodore . . . . .	27
Hardin, Joseph . . . . .	not available	Westerlund, Assar . . . . .	159
Icaza, Miguel de . . . . .	not available	Yocum, Ken . . . . .	109



## Message from the Program Chair

It's been my great pleasure to work with the FREENIX program committee in organizing this year's FREENIX track at the 1999 USENIX Annual Technical Conference. Although this is only the second time this track has been held at this conference, involvement has nonetheless increased rapidly. Many more papers than we could actually schedule were submitted this year, and attendees have shown a high degree of interest in the track. With 30 scheduled talks, a full schedule of evening BOFs, and a USENIX-sponsored \*BSD and Linux CD giveaway program for all attendees, it's a virtual certainty that no one will leave this year's conference ignorant of what the various free Unix (and free Unix application) alternatives have to offer!

Jordan Hubbard  
FREENIX Program Chair



## ERRATUM

Accidentally, an incomplete and preliminary version of the paper "A Future-Adaptable Password Scheme" by Niels Provos and David Mazières ended up in these *Proceedings*. **Please disregard the version of the paper herein.** Instead, read the version available at <http://www.usenix.org/events/usenix99/provos.html>.

When citing the paper, please use the following reference:

Niels Provos and David Mazières. A future-adaptable password scheme. In *FREENIX Track: 1999 USENIX Technical Conference* (the electronic version), June 1999. <http://www.usenix.org/events/usenix99/provos.html>.

# Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem

Marshall Kirk McKusick

*Author and Consultant*

Gregory R. Ganger

*Carnegie Mellon University*

## ABSTRACT

Traditionally, filesystem consistency has been maintained across system failures either by using synchronous writes to sequence dependent metadata updates or by using write-ahead logging to atomically group them. Soft updates, an alternative to these approaches, is an implementation mechanism that tracks and enforces metadata update dependencies to ensure that the disk image is always kept consistent. The use of soft updates obviates the need for a separate log or for most synchronous writes. Indeed, the ability of soft updates to aggregate many operations previously done individually and synchronously reduces the number of disk writes by 40 to 70% for file-intensive environments (e.g., program development, mail servers, etc.). In addition to performance enhancement, soft updates can also maintain better disk consistency. By ensuring that the only inconsistencies are unclaimed blocks or inodes, soft updates can eliminate the need to run a filesystem check program after every system crash. Instead, the system is brought up immediately. When it is convenient, a background task can be run on the active filesystem to reclaim any lost blocks and inodes.

This paper describes an implementation of soft updates and its incorporation into the 4.4BSD fast filesystem. It details the changes that were needed, both to the original research prototype and to the BSD system, to create a production-quality system. It also discusses the experiences, difficulties, and lessons learned in moving soft updates from research to reality; as is often the case, non-focal operations (e.g., **fsck** and “**fsync**”) required rethinking and additional code. Experiences with the resulting system validate the earlier research: soft updates integrates well with existing filesystems and enforces metadata dependencies with performance that is within a few percent of optimal.

## 1. Background and Introduction

In filesystems, **metadata** (e.g., directories, inodes and free block maps) gives structure to raw storage capacity. Metadata provides pointers and descriptions for linking multiple disk sectors into files and identifying those files. To be useful for persistent storage, a filesystem must maintain the integrity of its metadata in the face of unpredictable system crashes, such as power interruptions and operating system failures. Because such crashes usually result in the loss of all information in volatile main memory, the information in non-volatile storage (i.e., disk) must always be consistent enough to deterministically reconstruct a coherent filesystem state. Specifically, the on-disk

image of the filesystem must have no dangling pointers to uninitialized space, no ambiguous resource ownership caused by multiple pointers, and no unreferenced live resources. Maintaining these invariants generally requires sequencing (or atomic grouping) of updates to small on-disk metadata objects.

Traditionally, the BSD fast filesystem (FFS) [McKusick et al, 1984] and its derivatives have used synchronous writes to properly sequence stable storage changes. For example, creating a file in a BSD system involves first allocating and initializing a new inode and then filling in a new directory entry to point to it. With the synchronous write approach, the filesystem forces an application that creates a file to

wait for the disk write that initializes the on-disk inode. As a result, filesystem operations like file creation and deletion proceed at disk speeds rather than processor/memory speeds in these systems [McVoy & Kleiman, 1991; Ousterhout, 1990; Seltzer et al, 1993]. Since disk access times are long compared to the speeds of other computer components, synchronous writes reduce system performance.

The metadata update problem can also be addressed with other mechanisms. For example, one can eliminate the need to keep the on-disk state consistent by using NVRAM technologies, such as an uninterruptable power supply or Flash RAM [Wu & Zwaenepoel, 1994]. With this approach, only updates to the NVRAM need to be kept consistent, and updates can propagate to disk in any order and whenever it is convenient. Another approach is to group each set of dependent updates as an atomic operation with some form of write-ahead logging [Chutani et al, 1992; Hagmann, 1987; NCR\_Corporation, 1992] or shadow-paging [Chamberlin et al, 1981; Chao et al, 1992; Rosenblum & Ousterhout, 1991; Stonebraker, 1987]. Generally speaking, these approaches augment the on-disk state with additional information that can be used to reconstruct the committed metadata values after any system failure other than media corruption. Many modern filesystems successfully use write-ahead logging to improve performance compared to the synchronous write approach.

In [Ganger & Patt, 1994], an alternative approach called **soft updates** was proposed and evaluated in the context of a research prototype. With soft updates, the filesystem uses delayed writes (i.e., write-back caching) for metadata changes, tracks dependencies between updates, and enforces these dependencies at write-back time. Because most metadata blocks contain many pointers, cyclic dependencies occur frequently when dependencies are recorded only at the block level. Therefore, soft updates tracks dependencies on a per-pointer basis, which allows blocks to be written in any order. Any still-dependent updates in a metadata block are rolled-back before the block is written and rolled-forward afterwards. Thus, dependency cycles are eliminated as an issue. With soft updates, applications always see the most current copies of metadata blocks and the disk always sees copies that are consistent with its other contents.

In this paper, we describe the incorporation of soft updates into the 4.4BSD FFS used in the NetBSD, OpenBSD, FreeBSD, and BSDI operating systems. In doing so, we discuss experiences and lessons learned and describe the aspects that were more complex than was suggested in the original research papers. As is

often the case, non-focal operations like bounding the use of kernel memory used to track dependencies, fully implementing the “fsync”, system call semantics, properly detecting and handling lost resources in **fsck**, and cleanly and expediently completing an **unmount** system call required some rethinking and resulted in much of the code complexity. Despite these difficulties, our performance experiences do verify the conclusions of the early research. Specifically, using soft updates in BSD FFS eliminates most synchronous writes and provides performance that is within 5 percent of ideal (i.e., the same filesystem with no update ordering). At the same time, soft updates allows BSD FFS to provide cleaner semantics, stronger integrity and security guarantees, and immediate crash recovery (**fsck** not required for safe operation after a system crash).

The remainder of this paper is organized as follows. Section 2 describes the update dependencies that arise in BSD FFS operations. Section 3 describes how the BSD soft updates implementation handles these update dependencies, including the key data structures, how they are used, and how they are incorporated into the 4.4BSD operating system. Section 4 discusses experiences and lessons learned from converting a prototype implementation into an implementation suitable for use in production environments. Section 5 briefly overviews performance results from 4.4BSD systems using soft updates. Section 6 discusses new support for filesystem snapshots and how this support can be used to do a partial **fsck** in the background on active filesystems. Section 7 outlines the status and availability of the BSD soft updates code.

## 2. Update Dependencies in the BSD Fast Filesystem

Several important filesystem operations consist of a series of related modifications to separate metadata structures. To ensure recoverability in the presence of unpredictable failures, the modifications often must be propagated to stable storage in a specific order. For example, when creating a new file, the filesystem allocates an inode, initializes it and constructs a directory entry that points to it. If the system goes down after the new directory entry has been written to disk but before the initialized inode is written, consistency may be compromised since the contents of the on-disk inode are unknown. To ensure metadata consistency, the initialized inode must reach stable storage before the new directory entry. We refer to this requirement as an **update dependency**, because safely writing the directory entry depends on first writing the inode. The ordering constraints map onto three simple rules:

- 1) Never point to a structure before it has been initialized (e.g., an inode must be initialized before a directory entry references it).
- 2) Never re-use a resource before nullifying all previous pointers to it (e.g., an inode's pointer to a data block must be nullified before that disk block may be re-allocated for a new inode).
- 3) Never reset the old pointer to a live resource before the new pointer has been set (e.g., when renaming a file, do not remove the old name for an inode until after the new name has been written).

This section describes the update dependencies that arise in BSD FFS, assuming a basic understanding of BSD FFS as described in [McKusick et al, 1996]. There are eight BSD FFS activities that require update ordering to ensure post-crash recoverability: file creation, file removal, directory creation, directory removal, file/directory rename, block allocation, indirect block manipulation, and free map management.

The two main resources managed by the BSD FFS are inodes and data blocks. Two bitmaps are used to maintain allocation information about these resources. For each inode in the filesystem, the inode bitmap has a bit that is set if the inode is in use and cleared if it is free. For each block in the filesystem, the data block bitmap has a bit that is set if the block is free and cleared if it is in use. Each FFS filesystem is broken down into fixed-size pieces referred to as cylinder groups. Each cylinder group has a cylinder group block that contains the bitmaps for the inodes and data blocks residing within that cylinder group. For a large filesystem, this organization allows just those sub-pieces of the filesystem bitmap that are actively being used to be brought into the kernel memory. Each of these active cylinder group blocks is stored in a separate I/O buffer and can be written to disk independently of the other cylinder group blocks.

When a file is created, three metadata structures located in separate blocks are modified. The first is a new inode, which is initialized with its type field set to the new file type, its link count set to one to show that it is live (i.e., referenced by some directory), its permission fields set as specified, and all other fields set to default values. The second is the inode bitmap, which is modified to show that the inode has been allocated. The third is a new directory entry, which is filled in with the new name and a pointer to the new inode. To ensure that the bitmaps always reflect all allocated resources, the bitmap must be written to disk before the inode or directory entry. Because the inode is in an unknown state until after it has been initialized on the disk, rule #1 specifies that there is an update

dependency requiring that the relevant inode be written before the relevant directory entry. Although not strictly necessary, most BSD fast filesystem implementations also immediately write the directory block before the system call creating the file returns. This second synchronous write ensures that the filename is on stable storage if the application later does an "fsync" system call. If it were not done, then the "fsync" call would have to be able to find all the unwritten directory blocks containing a name for the file and write them to disk. A similar update dependency between inode and directory entry exists when adding a second name for the same file (a.k.a. a hard link), since the addition of the second name requires the filesystem to increment the link count in the inode and write the inode to disk before the entry may appear in the directory.

When a file is deleted, a directory block, an inode block, and one or more cylinder group bitmaps are modified. In the directory block, the relevant directory entry is "removed", usually by nullifying the inode pointer. In the inode block, the relevant inode's type field, link count and block pointers are zeroed out. The deleted file's blocks and inode are then added to the appropriate free block/inode maps. Rule #2 specifies that there are update dependencies between the directory entry and the inode and between the inode and any modified free map bits. To keep the link count conservatively high (and reduce complexity in practice), the update dependency between a directory entry and inode also exist when removing one of multiple names (hard links) for a file.

Creation and removal of directories is largely as described above for regular files. However, the "." entry is a link from the child directory to the parent, which adds additional update dependencies. Specifically, during creation, the parent's link count must be incremented on disk before the new directory's "." pointer is written. Likewise, during removal, the parent's link count must be decremented after the removed directory's "." pointer is nullified. (Note that this nullification is implicit in deleting the child directory's pointer to the corresponding directory block.)

When a new block is allocated, its bitmap location is updated to reflect that it is in use and the block's contents are initialized with newly written data or zeros. In addition, a pointer to the new block is added to an inode or indirect block (see below). To ensure that the on-disk bitmap always reflects allocated resources, the bitmap must be written to disk before the pointer. Also, because the contents of the newly allocated disk location are unknown, rule #1 specifies an update

dependency between the new block and the pointer to it. Because enforcing this update dependency with synchronous writes can reduce data creation throughput by a factor of two [Ganger & Patt, 1994], many implementations ignore it for regular data blocks. This implementation decision reduces integrity and security, since newly allocated blocks generally contain previously deleted file data. Soft updates allows all block allocations to be protected in this way with near-zero performance reduction.

Manipulation of indirect blocks does not introduce fundamentally different update dependencies, but they do merit separate discussion. Allocation, both of indirect blocks and of blocks pointed to by indirect blocks, is as discussed above. File deletion, and specifically de-allocation, is more interesting for indirect blocks. Because the inode reference is the only way to identify indirect blocks and blocks connected to them (directly or indirectly), nullifying the inode's pointer to an indirect block is enough to eliminate all recoverable pointers to said blocks. Once the pointer is nullified on disk, all its blocks can be freed. Only for partial truncation of a file do update dependencies between indirect block pointers and the pointed-to blocks exist. Some FFS implementations do not exploit this distinction, even though it can have a significant effect on the time required to remove a large file.

When a file is being renamed, two directory entries are affected. A new entry (with the new name) is created and set to point to the relevant inode and the old entry is removed. Rule #3 states that the new entry should be written to disk before the old entry is removed to avoid having the file unreferenced on reboot. If link counts are being kept conservatively, rename involves at least four disk updates in sequence: one to increment the inode's link count, one to add the new directory entry, one to remove the old directory entry, and one to decrement the link count. If the new name already existed, then the addition of the new directory entry also acts as the first step of file removal as discussed above. Interestingly, rename is the one POSIX file operation that really wants an atomic update to multiple user-visible metadata structures to provide ideal semantics. POSIX does not require said semantics and most implementations cannot provide it.

On an active filesystem, the bitmaps change constantly. Thus, the copy of the bitmaps in the kernel memory often differs from the copy that is stored on the disk. If the system halts without writing out the incore state of the bitmaps, some of the recently allocated inodes and data blocks may not be reflected in the out-of-date copies of the bitmaps on the disk. So, the filesystem check program, **fsck**, must be run over all the inodes in

the filesystem to ascertain which inodes and blocks are in use and bring the bitmaps up to date [McKusick & Kowalski, 1994]. An added benefit of soft updates is that it tracks the writing of the bitmaps to disk and uses this information to ensure that no newly allocated inodes or pointers to newly allocated data blocks will be written to disk until after the bitmap that references them has been written to disk. This guarantee ensures that there will never be an allocated inode or data block that is not marked in the on-disk bitmap. This guarantee, together with the other guarantees made by the soft update code, means that it is no longer necessary to run **fsck** after a system crash. This feature is discussed further in Section 6.

### 3. Tracking and Enforcing Update Dependencies

This section describes the BSD soft updates data structures and their use in enforcing the update dependencies described in Section 2. The structures and algorithms described eliminate all synchronous write operations from BSD FFS except for the partial truncation of a file and the "fsync" system call, which explicitly requires that all the state of a file be committed to disk before the system call returns.

The key attribute of soft updates is dependency tracking at the level of individual changes within cached blocks. Thus, for a block containing 64 inodes, the system can maintain up to 64 dependency structures with one for each inode in the buffer. Similarly for a buffer containing a directory block containing 50 names, the system can maintain up to 50 dependency structures with one for each name in the directory. With this level of detailed dependency information, circular dependencies between blocks are not problematic. For example, when the system wishes to write a buffer containing inodes, those inodes that can be safely written can go to the disk. Any inodes that cannot be safely written yet are temporarily rolled back to their safe values while the disk write proceeds. After the disk write completes, such inodes are rolled forward to their current values. Because the buffer is locked throughout the time that the contents are rolled back, the disk write is being done, and the contents are rolled forward, any processes wishing to use the buffer will be blocked from accessing it until it has been returned to its current state.

#### 3.1. Dependency Structures

A soft updates implementation uses a variety of data structures to track pending update dependencies among filesystem structures. Table 1 lists the dependency structures used in the BSD soft updates implementation, their main functions, and the types of

Name	Function	Associated Structures
bmsafemap	track bitmap dependencies (points to lists of dependency structures for recently allocated blocks and inodes)	cylinder group block
inodedep	track inode dependencies (information and list head pointers for all inode-related dependencies, including changes to the link count, the block pointers, and the file size)	inode block
allocdirect	track inode-referenced block (linked into lists pointed to by an inodedep and a bmsafemap to track inode's dependence on the block and bitmap being written to disk)	data block or indirect block or directory block
indirdep	track indirect block dependencies (points to list of dependency structures for recently-allocated blocks with pointers in the indirect block)	indirect block
allocindir	track indirect block-referenced block (linked into lists pointed to by an indirdep and a bmsafemap to track the indirect block's dependence on that block and bitmap being written to disk)	data block or indirect block or directory block
pagedep	track directory block dependencies (points to lists of diradd and dirrem structures)	directory block
diradd	track dependency between a new directory entry and the referenced inode	inodedep and directory block
mkdir	track new directory creation (used in addition to standard diradd structure when doing a mkdir)	inodedep and directory block
dirrem	track dependency between a deleted directory entry and the unlinked inode	first pagedep then tasklist
freefrag	tracks a single block or fragment to be freed as soon as the corresponding block (containing the inode with the now-replaced pointer to it) is written to disk	first inodedep then tasklist
freeblks	tracks all the block pointers to be freed as soon as the corresponding block (containing the inode with the now-zeroed pointers to them) is written to disk	first inodedep then tasklist
freefile	tracks the inode that should be freed as soon as the corresponding block (containing the inode block with the now-reset inode) is written to disk	first inodedep then tasklist

**Table 1: Soft Updates and Dependency Tracking**

blocks with which they can be associated. These dependency structures are allocated and associated with blocks as various file operations are completed. They are connected to the in-core blocks with which they are associated by a pointer in the corresponding buffer header. Two common aspects of all listed dependency structures are the *worklist* structure and the states used to track the progress of a dependency.

The *worklist* structure is really just a common header included as the first item in each dependency structure. It contains a set of linkage pointers and a type field to show the type of structure in which it is embedded. The *worklist* structure allows several different types of dependency structures to be linked together into a single list. The soft updates code can traverse one of these heterogeneous lists, using the type field to determine which kind of dependency structure it has encountered, and take the appropriate action with each.

The typical use for the *worklist* structure is to link together a set of dependencies associated with a buffer. Each buffer in the system has a *worklist* head pointer added to it. Any dependencies associated with that

buffer are linked onto its *worklist* list. After the buffer has been locked and just before the buffer is to be written, the I/O system passes the buffer to the soft update code to let it know that a disk write is about to be initiated. The soft update code then traverses the list of dependencies associated with the buffer and does any needed roll-back operations. After the disk write completes but before the buffer is unlocked, the I/O system calls the soft update code to let it know that a write has completed. The soft update code then traverses the list of dependencies associated with the buffer, does any needed roll-forward operations, and deallocates any dependencies that are fulfilled by the data in the buffer having been written to disk.

Another important list maintained by the soft updates code is the *tasklist* that contains background tasks for the work daemon. Dependency structures are generally added to the *tasklist* during the disk write completion routine, describing tasks that have become safe given the disk update but that may need to block for locks or I/O and therefore cannot be completed during the interrupt handler. Once per second, the syncer daemon (in its dual role as the soft updates work daemon)

wakes up and calls into the soft updates code to process any items on the *tasklist*. The work done for a dependency structure on this list is type-dependent. For example, for a *freeblks* structure, the listed blocks are marked free in the block bitmaps. For a *dirrem* structure, the associated inode's link count is decremented, possibly triggering file deletion.

**Dependency States.** Most dependency structures have a set of flags that describe the state of completion of the corresponding dependency. Dirty cache blocks can be written to the disk at any time. When the I/O system hands the buffer to the soft updates code (before and after a disk write), the states of the associated dependency structures determine what actions are taken. Although the specific meanings vary from structure to structure, the three main flags and their general meanings are:

#### ATTACHED

The ATTACHED flag indicates that the buffer with which the dependency structure is associated is not currently being written. When a disk write is initiated for a buffer with a dependency that must be rolled-back, the ATTACHED flag is cleared in the dependency structure to show that it has been rolled-back in the buffer. When the disk write completes, updates described by dependency structures that have the ATTACHED flag cleared are rolled-forward and the ATTACHED flag is set. Thus, a dependency structure can never be deleted while its ATTACHED flag is cleared, since the information needed to do the roll-forward operation would then be lost.

#### DEPCOMPLETE

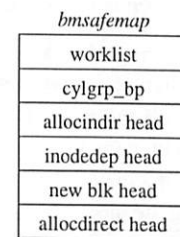
The DEPCOMPLETE flag indicates that all associated dependencies have been completed. When a disk write is initiated, the update described by a dependency structure is rolled-back if the DEPCOMPLETE flag is clear. For example, in a dependency structure that is associated with newly allocated inodes or data blocks, the DEPCOMPLETE flag is set when the corresponding bitmap has been written to disk.

#### COMPLETE

The COMPLETE flag indicates that the update being tracked has been committed to the disk. For some dependencies, updates will be rolled back during disk writes when the COMPLETE flag is clear. For example, for a newly allocated data block, the COMPLETE flag is set when the contents of the block have been written to disk.

In general, the flags are set as disk writes complete and a dependency structure can be deallocated only when its ATTACHED, DEPCOMPLETE, and COMPLETE flags are all set. Consider the example of a newly allocated data block that will be tracked by an *allocdirect* structure. The ATTACHED flag will initially be set when the allocation occurs. The DEPCOMPLETE flag will be set after the bitmap allocating that new block is written. The COMPLETE flag will be set after the contents of the new block are written. If the inode claiming the newly allocated block is written before both the DEPCOMPLETE and COMPLETE flags are set, the ATTACHED flag will be cleared while the block pointer in the inode is rolled back to zero, the inode is written, and the block pointer in the inode is rolled forward to the new block number. Where different, the specific meanings of these flags in the various dependency structures are described in the subsections that follow.

### 3.2. Bitmap Dependency Tracking



**Figure 1:** Bitmap Update Dependencies

Bitmap updates are tracked by the *bmsafemap* structure shown in Figure 1. Each buffer containing a cylinder group block will have its own *bmsafemap* structure. As with every dependency structure, the first entry in the *bmsafemap* structure is a *worklist* structure. Each time an inode, direct block, or indirect block is allocated from the cylinder group, a dependency structure is created for that resource and linked onto the appropriate *bmsafemap* list. Each newly allocated inode will be represented by an *inodedep* structure linked to the *bmsafemap inodedep head* list. Each newly allocated block directly referenced by an inode will be represented by an *allocdirect* structure linked to the *bmsafemap allocdirect head* list. Each newly allocated block referenced by an indirect block will be represented by an *allocindir* structure linked to the *bmsafemap allocindir head* list. Because of the FFS code's organization, there is a small window between the time a block is first allocated and the time at which its use is known. During this period of time, it is described by a *newblk* structure linked to the *bmsafemap new blk head* list. After the kernel

chooses to write the cylinder group block, the soft update code will be notified when the write has completed. At that time, the code traverses the inode, direct block, indirect block, and new block lists, setting the DEPCOMPLETE flag in each dependency structure and removing said dependency structure from its dependency list. Having cleared all its dependency lists, the *bmsafemap* structure can be deallocated. There are multiple lists as it is slightly faster and more type-safe to have lists of specific types.

### 3.3. Inode Dependency Tracking

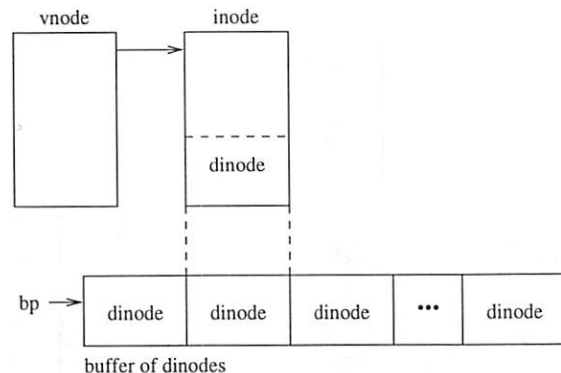
<i>inodedep</i>
worklist
state (see below)
deps list
dep bp
hash list
filesystem ptr
inode number
nlink delta
saved inode ptr
saved size
pending ops head
buf wait head
inode wait head
buffer update head
incore update head

**Figure 2: Inode Update Dependencies**

Inode updates are tracked by the *inodedep* structure shown in Figure 2. The *worklist* and “state” fields are as described for dependency structures in general. The “filesystem ptr” and “inode number” fields identify the inode in question. When an inode is newly allocated, its *inodedep* is attached to the “inodedep head” list of a *bmsafemap* structure. Here, “deps list” chains additional new *inodedeps* and “dep bp” points to the cylinder group block that contains the corresponding bitmap. Other *inodedep* fields are explained in subsequent subsections.

Before detailing the rest of the dependencies associated with an inode, we need to describe the steps involved in updating an inode on disk as pictured in Figure 3.

Step 1: The kernel calls the *vnode* operation, *VOP\_UPDATE*, which requests that the disk resident part of an inode (referred to as a *dinode*) be copied from its in-memory *vnode* structure to the appropriate disk buffer. This



**Figure 3: Inode Update Steps**

disk buffer holds the contents of an entire disk block, which is usually big enough to include 64 *dinodes*. Some dependencies are fulfilled only when the inode has been written to disk. For these, dependency structures need to track the progress of the writing of the inode. Therefore, during step 1, a soft update routine, “*softdep\_update\_inodeblock*”, is called to move *allocdirect* structures from the “incore update” list to the “buffer update” list and to move *freefile*, *freeblks*, *freefrag*, *diradd*, and *mkdir* structures (described below) from the “inode wait” list to the “buf wait” list.

Step 2: The kernel calls the *vnode* operation, *VOP\_STRATEGY*, which prepares to write the buffer containing the *dinode*, pointed to by *bp* in Figure 3. A soft updates routine, “*softdep\_disk\_io\_initiation*”, identifies *inodedep* dependencies and calls “*initiate\_write\_inodeblock*” to do roll-backs as necessary.

Step 3: Output completes on the buffer referred to by *bp* and the I/O system calls a routine, “*biodone*”, to notify any waiting processes that the write has finished. The “*biodone*” routine then calls a soft updates routine, “*softdep\_disk\_write\_complete*”, which identifies *inodedep* dependencies and calls “*handle\_written\_inodeblock*” to revert roll-backs and clear any dependencies on the “buf wait” and “buffer update” lists.

### 3.4. Direct Block Dependency Tracking

Figure 4 illustrates the dependency structures involved in allocation of direct blocks. Recall that the key dependencies are that, before the on-disk inode points

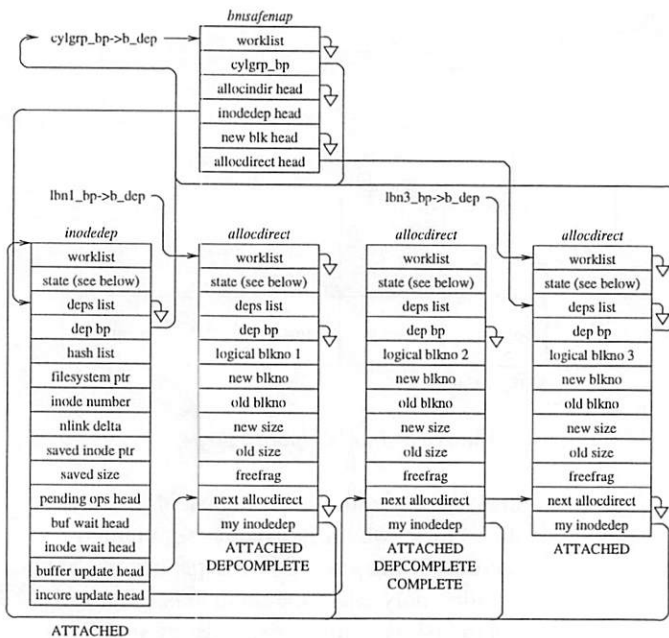


Figure 4: Direct Block Allocation Dependencies

to a newly allocated block, both the corresponding bitmap block and the new block itself must be written to the disk. The order in which the two dependencies complete is not important. The figure introduces the *allocdirect* structure which tracks blocks directly referenced by the inode. The three recently allocated logical blocks (1, 2, and 3) shown are each in a different state. For logical block 1, the bitmap block dependency is complete (as shown by the DEPCOMPLETE flag being set), but the block itself has not yet been written (as shown by the COMPLETE flag being cleared). For logical block 2, both dependencies are complete. For logical block 3, neither dependency is complete, so the corresponding *allocdirect* structure is attached to a *bmsafemap* "allocdirect head" list (recall that this list is traversed to set DEPCOMPLETE flags after bitmap blocks are written). The COMPLETE flag for logical blocks 1 and 3 will be set when their initialized data blocks are written to disk. The figure also shows that logical block 1 existed at a time that VOP\_UPDATE was called, which is why its *allocdirect* structure resides on the *inodedep* "buffer update" list. Logical blocks 2 and 3 were created after the most recent call to VOP\_UPDATE and thus their structures reside on the *inodedep* "incore update" list.

For files that grow in small steps, a direct block pointer may first point to a fragment that is later promoted to a larger fragment and eventually to a full-sized block. When a fragment is replaced by a larger fragment or a full-sized block, it must be released back to the

filesystem. However, it cannot be released until the new fragment or block has had its bitmap entry and contents written and the inode claiming the new fragment or block has been written to the disk. The fragment to be released is described by a *freefrag* structure (not shown). The *freefrag* structure is held on the "freefrag" list of the *allocdirect* for the block that will replace it until the new block has had its bitmap entry and contents written. The *freefrag* structure is then moved to the "inode wait" list of the *inodedep* associated with its *allocdirect* structure where it migrates to the "buf wait" list when VOP\_UPDATE is called. The *freefrag* structure eventually is added to the *tasklist* after the buffer holding the inode block has been written to disk. When the *tasklist* is serviced, the fragment listed in the *freefrag* structure is returned to the free-block bitmap.

### 3.5. Indirect Block Dependency Tracking

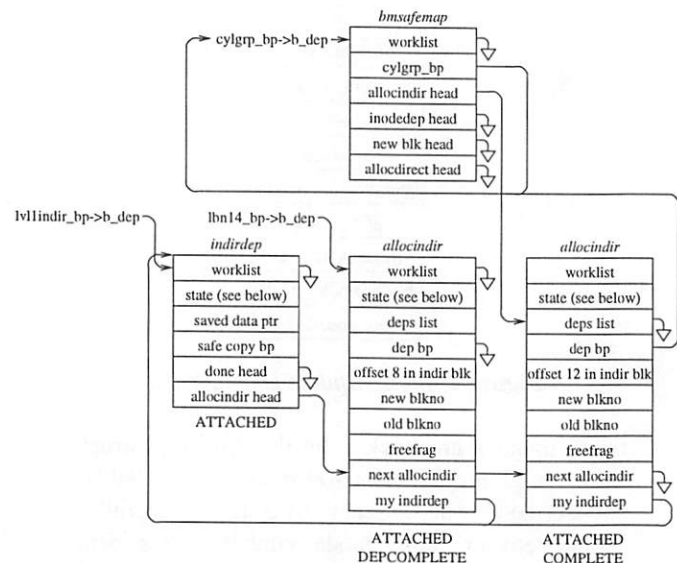
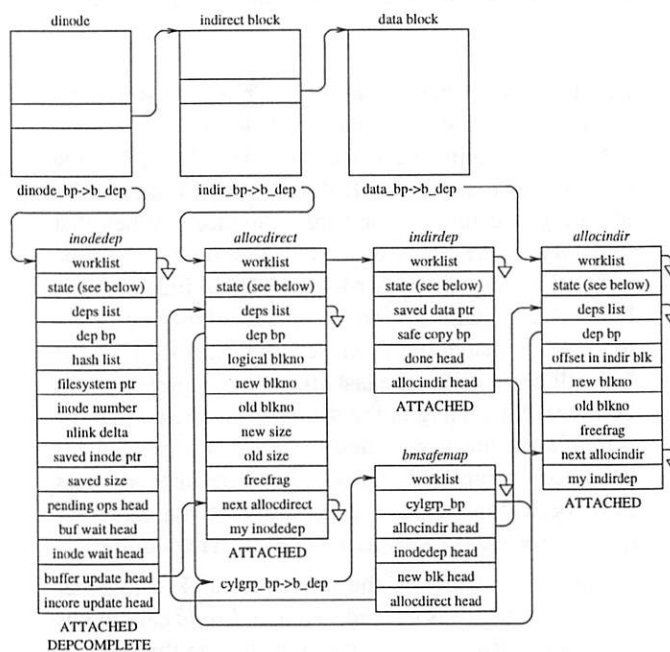


Figure 5: Indirect Block Allocation Dependencies

Figure 5 shows the dependency structures involved in allocation of indirect blocks, which includes the same dependencies as with direct blocks. This figure introduces two new dependency structures. A separate *allocindir* structure tracks each individual block pointer in an indirect block. The *indirdep* structure manages all the *allocindir* dependencies associated with an indirect block. The figure shows a file that recently allocated logical blocks 14 and 15 (the third and fourth entries, at offsets 8 and 12, in the first indirect block). The allocation bitmaps have been written for logical block 14 (as shown by its DEPCOMPLETE flag being set), but not for block 15. Thus, the

*bmsafemap* structure tracks the *allocindir* structure for logical block 15. The contents of logical block 15 have been written to disk (as shown by its COMPLETE flag being set), but not those of block 14. The COMPLETE flag will be set in 14's *allocindir* structure once the block is written. The list of *allocindir* structures tracked by an *indirdep* structure can be quite long (e.g., up to 2048 entries for 8KB indirect blocks). To avoid traversing lengthy dependency structure lists in the I/O routines, an *indirdep* structure maintains a second version of the indirect block: the "saved data ptr" always points to the buffer's up-to-date copy and the "safe copy ptr" points to a version that includes only the subset of pointers that can be safely written to disk (and NULL for the others). The former is used for all filesystem operations and the latter is used for disk writes. When the "allocindir head" list becomes empty, the "saved data ptr" and "safe copy ptr" point to identical blocks and the *indirdep* structure (and the safe copy) can be deallocated.

### 3.6. Dependency Tracking for new Indirect Blocks

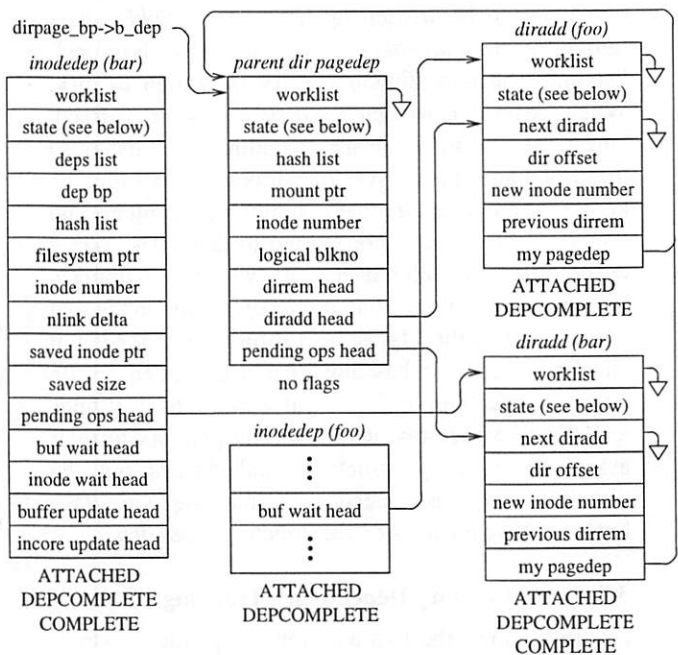


**Figure 6:** Dependencies for a File Expanding into an Indirect Block

Figure 6 shows the structures associated with a file that recently expanded into its single-level indirect block. Specifically, this involves *inodedep* and *indirdep* structures to manage dependency structures for the inode and indirect block, an *allocdirect* structure to track the dependencies associated with the indirect block's allocation, and an *allocindir* structure to track the

dependencies associated with a newly allocated block pointed to by the indirect block. These structures are used as described in the previous three subsections. Neither the indirect block nor the data block that it references have had their bitmaps set, so they do not have their DEPCOMPLETE flag set and are tracked by a *bmsafemap* structure. The bitmap entry for the inode has been written, so the *inodedep* structure has its DEPCOMPLETE flag set. The use of the "buffer update head" list by the *inodedep* structure indicates that the in-core inode has been copied into its buffer by a call to VOP\_UPDATE. Neither of the dependent pointers (from the inode to the indirect block and from the indirect block to the data block) can be safely included in disk writes yet, since the corresponding COMPLETE and DEPCOMPLETE flags are not set. Only after the bitmaps and the contents have been written will all the flags be set and the dependencies complete.

### 3.7. New Directory Entry Dependency Tracking



**Figure 7:** Dependencies Associated with Adding New Directory Entries

Figure 7 shows the dependency structures for a directory that has two new entries, **foo** and **bar**. This figure introduces two new dependency structures. A separate *diradd* structure tracks each individual directory entry in a directory block. The *pagedep* structure manages all the *diradd* dependencies associated with a directory block. For each new file, there is an *inodedep* structure and a *diradd* structure. Both files' inodes have had

their bitmap's written to disk, as shown by the DEP-COMplete flags being set in their *inodedeps*. The inode for **foo** has been updated with VOP\_UPDATE, but has not yet been written to disk, as shown by the COMPLETE flag on its *inodedep* structure not being set and by its *diradd* structure still being linked onto its "buf wait" list. Until the inode is written to disk with its increased link count, the directory entry may not appear on disk. If the directory page is written, the soft updates code will roll back the creation of the new directory entry for **foo** by setting its inode number to zero. After the disk write completes, the roll-back is reversed by restoring the correct inode number for **foo**.

The inode for **bar** has been written to disk, as shown by the COMPLETE flag being set in its *inodedep* and *diradd* structures. When the inode write completed, the *diradd* structure for **bar** was moved from the *inodedep* "buf wait" list to the *inodedep* "pending ops" list. The *diradd* also moved from the *pagedep* "diradd" list to the *pagedep* "pending ops" list. Since the inode has been written, it is safe to allow the directory entry to be written to disk. The *diradd* entries remain on the *inodedep* and *pagedep* "pending ops" list until the new directory entry is written to disk. When the entry is written, the *diradd* structure is freed. One reason to maintain the "pending ops" list is so that when an "fsync" system call is done on a file, the kernel is able to ensure that both the file's contents and directory reference(s) are written to disk. The kernel ensures that the reference(s) are written by doing a lookup to see if there is an *inodedep* for the inode that is the target of the "fsync". If it finds an *inodedep*, it checks to see if it has any *diradd* dependencies on either its "pending ops" or "buf wait" lists. If it finds any *diradd* structures, it follows the pointers to their associated *pagedep* structures and flushes out the directory inode associated with that *pagedep*. This back-tracking recurses on the directory *inodedep*.

### 3.8. New Directory Dependency Tracking

Figure 8 shows the two additional dependency structures involved with creating a new directory. For a regular file, the directory entry can be committed as soon as the newly referenced inode has been written to disk with its increased link count. When a new directory is created, there are two additional dependencies: writing the directory data block containing the "." and ".." entries (MKDIR\_BODY) and writing the parent inode with the increased link count for ".." (MKDIR\_PARENT). These additional dependencies are tracked by two *mkdir* structures linked to the associated *diradd* structure. The BSD soft updates design dictates that any given dependency will correspond to a single

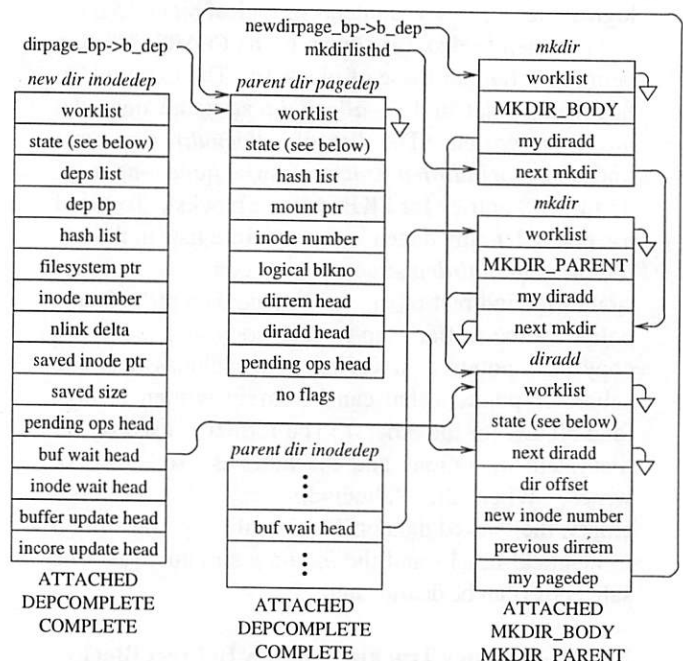
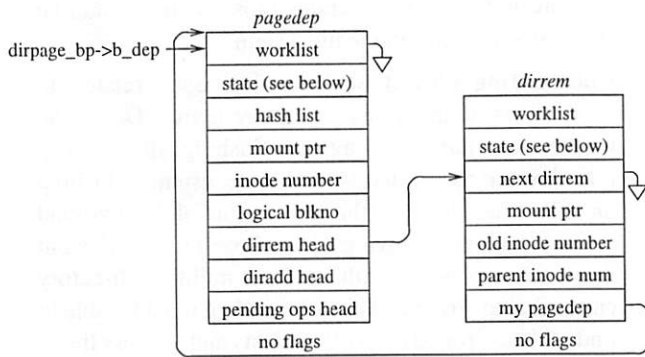


Figure 8: Dependencies Associated with Adding a New Directory

buffer at any given point in time. Thus, two structures are used to track the action of the two different buffers. When each completes, it clears its associated flag in the *diradd* structure. The MKDIR\_PARENT is linked to the *inodedep* structure for the parent directory. When that directory inode is written, the link count will be updated on disk. The MKDIR\_BODY is linked to the buffer that contains the initial contents of the new directory. When that buffer is written, the entries for "." and ".." will be on disk. The last *mkdir* to complete sets the DEP-COMplete flag in the *diradd* structure so that the *diradd* structure knows that these extra dependencies have been completed. Once these extra dependencies have been completed, the handling of the directory *diradd* proceeds exactly as it would for a regular file.

All *mkdir* structures in the system are linked together on a list. This list is needed so that a *diradd* can find its associated *mkdir* structures and deallocate them if it is prematurely freed (e.g., if a "mkdir" system call is immediately followed by a "rmdir" system call of the same directory). Here, the de-allocation of a *diradd* structure must traverse the list to find the associated *mkdir* structures that reference it. The deletion would be faster if the *diradd* structure were simply augmented to have two pointers that referenced the associated *mkdir* structures. However, these extra pointers would double the size of the *diradd* structure to speed an infrequent operation.

### 3.9. Directory Entry Removal Dependency Tracking



**Figure 9:** Dependencies Associated with Removing a Directory Entry

Figure 9 shows the dependency structures involved with the removal of a directory entry. This figure introduces one new dependency structure, the *dirrem* structure, and a new use for the *pagedep* structure. A separate *dirrem* structure tracks each individual directory entry to be removed in a directory block. In addition to previously described uses, *pagedep* structures associated with a directory block manage all *dirrem* structures associated with the block. After the directory block is written to disk, the *dirrem* request is added to the work daemon's *tasklist* list. For file deletions, the work daemon will decrement the inode's link count by one. For directory deletions, the work daemon will decrement the inode's link count by two, truncate its to size zero, and decrement the parent directory's link count by one. If the inode's link count drops to zero, the resource reclamation activities described in Section 3.11 are initiated.

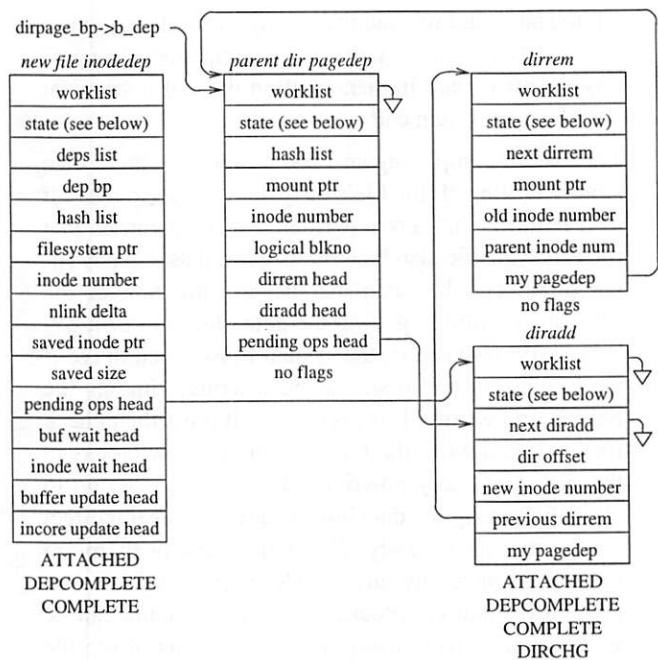
### 3.10. File Truncation

In the non-soft-updates FFS, when a file is truncated to zero length, the block pointers in its inode are saved in a temporary list, the pointers in the inode are zeroed, and the inode is synchronously written to disk. When the inode write completes, the list of its formerly claimed blocks are added to the free-block bitmap. With soft updates, the block pointers in the inode being truncated are copied into a *freeblks* structure, the pointers in the inode are zeroed, and the inode is marked dirty. The *freeblks* structure is added to the "inode wait" list, and it migrates to the "buf wait" list when VOP\_UPDATE is called. The *freeblks* structure is eventually added to the *tasklist* after the buffer holding the inode block has been written to disk. When the *tasklist* is serviced, the blocks listed in the *freeblks* structure are returned to the free-block bitmap.

### 3.11. File and Directory Inode Reclamation

When the link count on a file or directory drops to zero, its inode is zeroed to indicate that it is no longer in use. In the non-soft-updates FFS, the zeroed inode is synchronously written to disk and the inode is marked as free in the bitmap. With soft updates, information about the inode to be freed is saved in a *freefile* structure. The *freefile* structure is added to the "inode wait" list, and it migrates to the "buf wait" list when VOP\_UPDATE is called. The *freefile* structure eventually is added to the *tasklist* after the buffer holding the inode block has been written to disk. When the *tasklist* is serviced, the inode listed in the *freefile* structure is returned to the free inode map.

### 3.12. Directory Entry Renaming Dependency Tracking



**Figure 10:** Dependencies Associated with Renaming a Directory Entry

Figure 10 shows the structures involved in renaming a file. The dependencies follow the same series of steps as those for adding a new file entry, with two variations. First, when a roll-back of an entry is needed because its inode has not yet been written to disk, the entry must be set back to the previous inode number rather than to zero. The previous inode number is stored in a *dirrem* structure. The DIRCHG flag is set in the *diradd* structure so that the roll-back code knows to use the old inode number stored in the *dirrem* structure. The second variation is that, after the modified

directory entry is written to disk, the *dirrem* structure is added to the work daemon's *tasklist* list so that the link count of the old inode will be decremented as described in Section 3.9.

#### 4. Experiences and Lessons Learned

This section describes various issues that arose in moving soft updates from research prototype to being a production-quality component of the 4.4BSD operating system. Some of these issues were evident shortcomings of the research prototype, and some were simply the result of differences in the host operating systems. Others, however, only became evident as we gained operational experience with soft updates.

**The "fsync" system call.** An important filesystem interface is accessed through the "fsync" system call. This system call requests that the specified file be written to stable storage and that the system call not return until all its associated writes have completed. The prototype soft update implementation did not implement the "fsync" system call.

The task of completing an "fsync" requires more than simply writing all the file's dirty data blocks to disk. It also requires that any unwritten directory entries that reference the file also be written, as well as any unwritten directories between the file and the root of the filesystem. Simply getting the data blocks to disk can be a major task. First, the system must check to see if the bitmap for the inode has been written, finding the bitmap and writing it if necessary. It must then check for, find, and write the bitmaps for any new blocks in the file. Next, any unwritten data blocks must go to disk. Following the data blocks, any first level indirect blocks that have newly allocated blocks in them are written, followed by any double indirect blocks, then triple level indirect blocks. Finally, the inode can be written which will ensure that the contents of the file are on stable store. Ensuring that all names for the file are also on stable store requires data structures that can determine whether there are any uncommitted names and if so, in which directories they occur. For each directory containing an uncommitted name, the soft update code must go through the same set of flush operations that it has just done on the file itself.

Although the "fsync" system call must ultimately be done synchronously, this does not mean that the flushing operations must each be done synchronously. Instead, whole sets of bitmaps or data blocks are pushed into the disk queue and the soft update code then waits for all the writes to complete. This approach is more efficient because it allows the disk

subsystem to sort all the write requests into the most efficient order for writing. Still, the "fsync" part of the soft update code generates most of the remaining synchronous writes in the filesystem.

**Unmounting filesystems.** Another issue related to "fsync" is unmounting of filesystems. Doing an **unmount** requires finding and flushing all the dirty files that are associated with the filesystem. Flushing the files may lead to the generation of background activity such as removing files whose reference count drops to zero as a result of their nullified directory entries being written. Thus, the system must be able to find all background activity requests and process them. Even on a quiescent filesystem, several iterations of file flushes followed by background activity may be required. The 4.4BSD FFS allows for forcible unmounts of filesystems which allows the unmount to take place while the filesystem is actively in use, which required additional support.

**Removing directories.** The prototype implementation oversimplified the sequencing of updates involved with removing a directory. Specifically, the prototype allowed the removal of the directory's name and the removal of its "." entry to proceed in parallel. This meant that a crash could leave the directory in place with its "." entry removed. Although **fsck** could be modified to repair this problem, it is not acceptable when **fsck** is bypassed during crash recovery.

For correct operation, a directory's "." entry should not be removed until after the directory is persistently unlinked. Correcting this in the soft updates code introduced a delay of up to two minutes between unlinked a directory and its really being deallocated (when the "." entry is removed). Until the directory's "." entry is really removed, the link count on its parent will not be decremented. Thus, when a user removed one or more directories, the the link count of their former parent still reflected their being present for several minutes. This delayed link count decrement not only caused some questions from users, but also caused some applications to break. For example, the "rmdir" system call will not remove a directory that has a link count over two. This restriction means that a directory that recently had directories removed from it cannot be removed until its former directories have been fully deleted.

To fix these link-count problems, the BSD soft updates implementation augments the inode "nlink" field with a new field called "effnlink". The "nlink" field is still stored as part of the on-disk metadata and reflects the true link count of the inode. The "effnlink" field is maintained only in kernel memory and reflects the final

value that the “nlink” field will reach once all its outstanding operations are completed. All interactions with user applications report the value of the “effnlink” field which results in the illusion that everything has happened immediately.

**Block Reallocation.** Because it was not done in System V Release 4 UNIX, the prototype system did not handle block reallocation. In the 4.4BSD FFS, the filesystem sometimes changes the on-disk locations of file blocks as a file grows to lay the file out more contiguously. Thus, a block that is initially assigned to a file may be replaced as the file grows larger. Although the prototype code was prepared to handle upgrades of fragments to full-sized blocks in the last block of a file, it was not prepared to have full-sized blocks reallocated at interior parts of inodes and indirect blocks.

**Memory used for Dependency Structures.** One concern with soft updates is the amount of memory consumed by the dependency structures. This problem was attacked on two fronts: memory efficiency and usage bounding.

The prototype implementation generally used two structures for each update dependency. One was associated with the data that needed to be written and one with the data that depended on the write. For example, each time a new block was allocated, new dependency structures were associated with the allocated disk block, the bitmap from which the block was allocated, and the inode claiming the new disk block. The structure associated with the inode was dependent on the other two being written. The 4.4BSD soft updates code uses a single dependency structure (associated with the disk block) to describe a block allocation. There is a single dependency structure associated with each bitmap and each inode, and all related block allocation structures are linked into lists headed by these structures. That is, one block allocation structure is linked into the allocated block’s list, the bitmap’s list, and the inode’s list. By constructing lists rather than using separate structures, the demand on memory was reduced by about 40 percent.

In daily operation, we have found that the additional dynamic memory load placed on the kernel memory allocation area is about equal to the amount of memory used by vnodes plus inodes. For a system with 1000 vnodes, the additional peak memory load is about 300KB. The one exception to this guideline occurs when large directory trees are removed. Here, the filesystem code can get arbitrarily far ahead of the on-disk state, causing the amount of memory dedicated to dependency structures to grow without bound. The 4.4BSD soft update code was modified to monitor the

memory load for this case and not allow it to grow past a tunable upper bound. When the bound is reached, new dependency structures can only be created at the rate at which old ones are retired. The effect of this limit is to slow down the rate of removal to the rate at which the disk updates can be done. While this restriction slows the rate at which soft updates can normally remove files, it is still considerably faster than the traditional synchronous write filesystem. In steady-state, the soft update remove algorithm requires about one disk write for each ten files removed while the traditional filesystem requires at least two writes for every file removed.

**The fsck Utility.** As with the dual tracking of the true and effective link count, the changes needed to **fsck** became evident through operational experience. In a non-soft-updates filesystem implementation, file removal happens within a few milliseconds. Thus, there is a short period of time between the directory entry being removed and the inode being deallocated. If the system crashes during a bulk tree removal operation, there are usually no inodes lacking references from directory entries, though in rare instances there may be one or two. By contrast, in a system running with soft updates, many seconds may elapse between the times when the directory entry is deleted and the inode is deallocated. If the system crashes during a bulk tree removal operation, there are usually tens to hundreds of inodes lacking references from directory entries. Historically, **fsck** placed any unreferenced inodes into the **lost+found** directory. This action is reasonable if the filesystem has been damaged because of disk failure which results in the loss of one or more directories. However, it results in the incorrect action of stuffing the **lost+found** directory full of partially deleted files when running with soft updates. Thus, the **fsck** program must be modified to check that a filesystem is running with soft updates and clear out, rather than saving, unreferenced inodes (unless it has determined that unexpected damage has occurred to the filesystem, in which case the files are saved in **lost+found**).

A peripheral benefit of soft updates is that **fsck** can trust the allocation information in the bitmaps. Thus, it only needs to check the subset of inodes in the filesystem that the bitmaps indicate are in use. Although some of the inodes marked “in use” may be free, none of those marked free will ever be in use.

**Partial File Truncation.** Although the common case for deallocation is for all data in a file to be deleted, the “truncate” system call allows applications to delete only part of a file. This creates slightly more complicated update dependencies, including the need to have

deallocation dependencies for indirect blocks and the need to consider partially deleted data blocks. Because it is so uncommon, neither the prototype nor the BSD soft updates implementation optimizes this case; the conventional synchronous write approach is used instead.

## 5. Performance

This paper gives only a cursory look at soft updates performance. For a detailed analysis, including comparisons with other techniques, see [Ganger, McKusick, & Patt, ].

We place the performance of BSD FFS with soft updates in context by comparing it to the default BSD FFS (referred to below as "normal"), which uses synchronous writes for update ordering, and BSD FFS mounted with the `O_ASYNC` option (referred to below as "asynchronous"), which ignores all update dependencies. In asynchronous mode, all metadata updates are converted into delayed writes (a delayed write is one in which the buffer is simply marked dirty, put on a least-recently-used list, and not written until needed for some other purpose). Thus, the `O_ASYNC` data provides an upper bound on the performance of an update ordering scheme in BSD FFS. As expected, we have found that soft updates eliminates almost all synchronous writes and usually allows BSD FFS to achieve performance with 5 percent of the upper bound. Compared to using synchronous writes, file creation and removal performance increases by factors of 2 and 20, respectively. Overall, 4.4BSD systems tend to require 40 percent fewer disk writes and complete tasks 25 percent more quickly than when using the default 4.4BSD fast filesystem implementation.

To provide a feeling for how the system performs in normal operation, we present measurements from three different system tasks. The first task is our "filesystem torture test". This consists of 1000 runs of the Andrew benchmark, 1000 copy and removes of `/etc` with randomly selected pauses of 0-60 seconds between each copy and remove, and 500 executions of the `find` application from `/` with randomly selected pauses of 100 seconds between each run. The run of the torture test compares as follows:

Filesystem Configuration	Disk Writes		Running Time
	Sync	Async	
Normal	1,459,147	487,031	27hr, 15min
Asynchronous	0	1,109,711	19hr, 43min
Soft Updates	6	1,113,686	19hr, 50min

The overall result is that asynchronous and soft updates require 42 percent fewer writes (with almost no synchronous writes) and have a 28 percent shorter

running time. This is particularly impressive when one considers that the finds and the pauses involve no update dependencies, and the Andrew benchmark is largely CPU bound.

The second test consists of building and installing the FreeBSD system. This task is a real-world example of a program development environment. The results are as follows:

Filesystem Configuration	Disk Writes		Running Time
	Sync	Async	
Normal	162,410	39,924	2hr, 12min
Asynchronous	0	38,262	1hr, 44min
Soft Updates	1124	48,850	1hr, 44min

The overall result is that soft updates require 75 percent fewer writes and has a 21 percent shorter running time. Although soft updates initiates 30 percent more writes than asynchronous, the two result in the same running time.

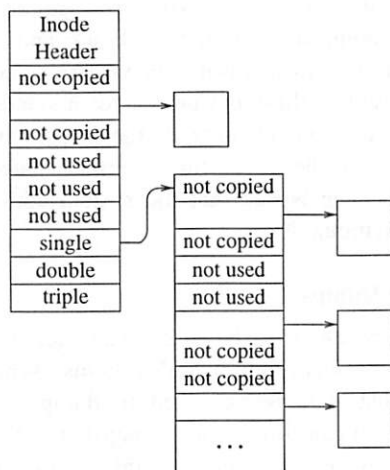
The third test compares the performance of the central mail server for Berkeley Software Design, Inc. run with and without soft updates. The administrator was obviously unwilling to run it in asynchronous mode, since it is a production machine and people will not abide by losing their mail. Unlike the tests above, which involve a single disk, the mail spool on this system is striped across three disks. The statistics were gathered by averaging the results from thirty days of non-weekend operation in each mode. The results for a 24-hour period are as follows:

Filesystem Configuration	Disk Writes	
	Sync	Async
Normal	1,877,794	1,613,465
Soft Updates	118,102	946,519

The normal filesystem averaged over 40 writes per second with a ratio of synchronous to asynchronous writes of 1:1. With soft updates, the write rate dropped to 12 per second and the ratio of synchronous to asynchronous writes dropped to 1:8. For this real-world application, soft updates requires 70 percent fewer writes, which triples the mail handling capacity of the machine.

## 6. Filesystem Snapshots

A filesystem **snapshot** is a frozen image of a filesystem at a given instant in time. Snapshots support several important features: the ability to provide back-ups of the filesystem at several times during the day, the ability to do reliable dumps of live filesystems, and (most important for soft updates) the ability to run a filesystem check program on a active system to reclaim lost blocks and inodes.



**Figure 11:** *Structure of a snapshot file*

Implementing snapshots in BSD FFS has proven to be straightforward, with the following steps. First, activity on the relevant filesystem is briefly suspended. Second, all system calls currently writing to that filesystem are allowed to finish. Third, the filesystem is synchronized to disk as if it were about to be unmounted. Finally, a **snapshot file** is created to track subsequent changes to the filesystem; a snapshot file is shown in Figure 11. This snapshot file is initialized to the size of the filesystem's partition, and most of its file block pointers are marked as "not copied". A few strategic blocks are allocated and copied, such as those holding copies of the superblock and cylinder group maps. The snapshot file also uses a distinguished block number (1) to mark all blocks "not used" at the time of the snapshot, since there is no need to copy those blocks if they are later allocated and written.

Once the snapshot file is in place, activity on the filesystem resumes. Each time an existing block in the filesystem is modified, the filesystem checks whether that block was in use at the time that the snapshot was taken (i.e., it is not marked "not used"). If so, and if it has not already been copied (i.e., it is still marked "not copied"), a new block is allocated and placed in the snapshot file to replace the "not copied" entry. The previous contents of the block are copied to the newly allocated snapshot file block, and the modification to the original is then allowed to proceed. Whenever a file is removed, the snapshot code inspects each of the blocks being freed and claims any that were in use at the time of the snapshot. Those blocks marked "not used" are returned to the free list.

When a snapshot file is read, reads of blocks marked "not copied" return the contents of the corresponding block in the filesystem. Reads of blocks that have been

copied return their contents. Writes to snapshot files are not permitted. When a snapshot file is no longer needed, it can be removed in the same way as any other file; its blocks are simply returned to the free list and its inode is zeroed and returned to the free inode list.

Snapshots may live across reboots. When a snapshot file is created, the inode number of the snapshot file is recorded in the superblock. When a filesystem is mounted, the snapshot list is traversed and all the listed snapshots are activated. The only limit on the number of snapshots that may exist in a filesystem is the size of the array in the superblock that holds the list of snapshots. Currently, this array can hold up to twenty snapshots.

Multiple snapshot files can concurrently exist. As described above, earlier snapshot files would appear in later snapshots. If an earlier snapshot is removed, a later snapshot would claim its blocks rather than allowing them to be returned to the free list. This semantic means that it would be impossible to free any space on the filesystem except by removing the newest snapshot. To avoid this problem, the snapshot code carefully goes through and expunges all earlier snapshots by changing its view of them to being zero length files. With this technique, the freeing of an earlier snapshot releases the space held by that snapshot.

When a block is overwritten, all snapshots are given an opportunity to copy the block. A copy of the block is made for each snapshot in which the block resides. Deleted blocks are handled differently. The list of snapshots is consulted. When a snapshot is found in which the block is active ("not copied"), the deleted block is claimed by that snapshot. The traversal of the snapshot list is then terminated. Other snapshots for which the block are active are left with an entry of "not copied" for that block. The result is that when they access that location they will still reference the deleted block. Since snapshots may not be written, the block will not change. Since the block is claimed by a snapshot, it will not be allocated to another use. If the snapshot claiming the deleted block is deleted, the remaining snapshots will be given the opportunity to claim the block. Only when none of the remaining snapshots want to claim the block (i.e., it is marked "not used" in all of them) will it be returned to the freelist.

### 6.1. Instant Filesystem Restart

Traditionally, after an unclean system shutdown, the filesystem check program, **fsck**, has had to be run over all inodes in an FFS filesystem to ascertain which inodes and blocks are in use and correct the bitmaps.

This is a painfully slow process that can delay the restart of a big server for an hour or more. The current implementation of soft updates guarantees the consistency of all filesystem resources, including the inode and block bitmaps. With soft updates, the only inconsistency that can arise in the filesystem (barring software bugs and media failures) is that some unreferenced blocks may not appear in the bitmaps and some inodes may have to have overly high link counts reduced. Thus, it is completely safe to begin using the filesystem after a crash without first running **fsck**. However, some filesystem space may be lost after each crash. Thus, there is value in having a version of **fsck** that can run in the background on an active filesystem to find and recover any lost blocks and adjust inodes with overly high link counts. A special case of the overly high link count is one that should be zero. Such an inode will be freed as part of reducing its link count to zero. This garbage collection task is less difficult than it might at first appear, since this version of **fsck** only needs to identify resources that are not in use and cannot be allocated or accessed by the running system.

With the addition of snapshots, the task becomes simple, requiring only minor modifications to the standard **fsck**. When run in background cleanup mode, **fsck** starts by taking a snapshot of the filesystem to be checked. **Fsck** then runs over the snapshot filesystem image doing its usual calculations just as in its normal operation. The only other change comes at the end of its run, when it wants to write out the updated versions of the bitmaps. Here, the modified **fsck** takes the set of blocks that it finds were in use at the time of the snapshot and removes this set from the set marked as in use at the time of the snapshot—the difference is the set of lost blocks. It also constructs the list of inodes whose counts need to be adjusted. **Fsck** then calls a new system call to notify the filesystem of the identified lost blocks so that it can replace them in its bitmaps. It also gives the set of inodes whose link counts need to be adjusted; those inodes whose link count is reduced to zero are truncated to zero length and freed. When **fsck** completes, it releases its snapshot.

## 6.2. User Visible Snapshots

Snapshots may be taken at any time. When taken every few hours during the day, they allow users to retrieve a file that they wrote several hours earlier and later deleted or overwrote by mistake. Snapshots are much more convenient to use than dump tapes and can be created much more frequently.

The snapshot described above creates a frozen image of a filesystem partition. To make that snapshot accessible to users through a traditional filesystem interface,

BSD uses the vnode driver, **vnd**. The **vnd** driver takes a file as input and produces a block and character device interface to access it. The **vnd** block device can then be used as the input device for a standard BSD FFS mount command, allowing the snapshot to appear as a replica of the frozen filesystem at whatever location in the namespace that the system administrator chooses to mount it.

## 6.3. Live Dumps

Once filesystem snapshots are available, it becomes possible to safely dump live filesystems. When **dump** notices that it is being asked to dump a mounted filesystem, it can simply take a snapshot of the filesystem and run over the snapshot instead of on the live filesystem. When **dump** completes, it releases the snapshot.

## 7. Current Status

The soft updates code is available for commercial use in Berkeley Software Design Inc.'s BSD/OS 4.0 and later systems. It is available for non-commercial use in the freely-available BSD systems: FreeBSD, NetBSD, and OpenBSD. The snapshot code is in alpha test and should be available in the BSD systems towards the end of 1999. Sun Microsystems has been evaluating the soft updates and snapshot technology for possible inclusion in Solaris. Vendors wishing to use soft updates for commercial use in a freely-available BSD or in their own products should visit <http://www.mckusick.com/softdep/> or contact Dr. McKusick.

## References

- Chamberlin et al, 1981.  
D. Chamberlin, M. Astrahan, & et al., "A History and Evaluation of System R," *Communications of the ACM*, 24, 10, p. 632–646 (1981).
- Chao et al, 1992.  
C. Chao, R. English, D. Jacobson, A. Stepanov, & J. Wilkes, *Mime: A High-Performance Parallel Storage Device with Strong Recovery Guarantees*, Hewlett-Packard Laboratories Report, HPL-CSP-92-9 rev 1 (November 1992).
- Chutani et al, 1992.  
S. Chutani, O. Anderson, M. Kazar, B. Leverett, W. Mason, & R. Sidebotham, "The Episode File System," *Winter USENIX Conference*, p. 43–60 (January 1992).
- Ganger, McKusick, & Patt, .  
G. Ganger, M. McKusick, & Y. Patt, "Soft Updates: A Solution to the Metadata Update Problem in Filesystems," *ACM Transactions on Computer Systems* (in preparation).

Ganger & Patt, 1994.

G. Ganger & Y. Patt, "Metadata Update Performance in File Systems," *USENIX Symposium on Operating Systems Design and Implementation*, p. 49–60 (November 1994).

Hagmann, 1987.

R. Hagmann, "Reimplementing the Cedar File System Using Logging and Group Commit," *ACM Symposium on Operating Systems Principles*, p. 155–162 (November 1987).

McKusick et al, 1996.

M. McKusick, K. Bostic, M. Karels, & J. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*, p. 269–271, Addison Wesley Publishing Company, Reading, MA (1996).

McKusick et al, 1984.

M. McKusick, W. Joy, S. Leffler, & R. Fabry, "A Fast File System for UNIX," *ACM Transactions on Computer Systems*, 2, 3, p. 181–197 (August 1984).

McKusick & Kowalski, 1994.

M. McKusick & T. Kowalski, "FSCK - The UNIX File System Check Program," *4.4 BSD System Manager's Manual*, p. 3:1–21, O'Reilly & Associates, Inc., Sebastopol, CA (1994).

McVoy & Kleiman, 1991.

L. McVoy & S. Kleiman, "Extent-like Performance From a UNIX File System," *Winter USENIX Conference*, p. 1–11 (January 1991).

NCR\_Corporation, 1992.

NCR\_Corporation, *Journaling File System Administrator Guide, Release 2.00*, NCR Document D1-2724-A (April 1992).

Ousterhout, 1990.

J. Ousterhout, "Why Aren't Operating Systems Getting faster As Fast As Hardware?," *Summer USENIX Conference*, p. 247–256 (June 1990).

Rosenblum & Ousterhout, 1991.

M. Rosenblum & J. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Symposium on Operating System Principles*, p. 1–15 (October 1991).

Seltzer et al, 1993.

M. Seltzer, K. Bostic, M. McKusick, & C. Staelin, "An Implementation of a Log-Structured File System for UNIX," *Winter USENIX Conference*, p. 201–220 (January 1993).

Stonebraker, 1987.

M. Stonebraker, "The Design of the POSTGRES Storage System," *Very Large DataBase Conference*, p. 289–300 (1987).

Wu & Zwaenepoel, 1994.

M. Wu & W. Zwaenepoel, "eNVy: A Non-Volatile, Main Memory Storage System," *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, p. 86–97 (October 1994).

## 8. Biographies

Dr. Marshall Kirk McKusick writes books and articles, consults, and teaches classes on UNIX- and BSD-related subjects. While at the University of California at Berkeley, he implemented the 4.2BSD fast filesystem, and was the Research Computer Scientist at the Berkeley Computer Systems Research Group (CSRG) overseeing the development and release of 4.3BSD and 4.4BSD. His particular areas of interest are the virtual-memory system and the filesystem. One day, he hopes to see them merged seamlessly. He earned his undergraduate degree in Electrical Engineering from Cornell University, and did his graduate work at the University of California at Berkeley, where he received Masters degrees in Computer Science and Business Administration, and a doctoral degree in Computer Science. He is a past president of the Usenix Association, and is a member of ACM and IEEE.

In his spare time, he enjoys swimming, scuba diving, and wine collecting. The wine is stored in a specially constructed wine cellar (accessible from the web at <http://www.mckusick.com/mckusick/>) in the basement of the house that he shares with Eric Allman, his domestic partner of 20-and-some-odd years. You can contact him via email at <mckusick@mckusick.com>.

Greg Ganger is an assistant professor of Electrical and Computer Engineering and Computer Science at Carnegie Mellon University. He has broad research interests in computer systems, including operating systems, networking, storage systems, computer architecture, performance evaluation and distributed systems. He also enjoys working on the occasional filesystem project. He spent 2+ years at MIT working on the exokernel operating system and related projects as part of the Parallel and Distributed Operating Systems group. He earned his various degrees (B.S, M.S, and Ph.D.) from the University of Michigan. He is a member of ACM and IEEE Computer Society. Greg never seems to have spare time, but does very much enjoy a good game of basketball. You can contact him via email at <ganger@ece.cmu.edu>.



# Design and Implementation of a Transaction-Based Filesystem on FreeBSD

Jason Evans

*The Hungry Programmers*

jasone@hungry.com, <http://www.hungry.com/~jasone>

## Abstract

Transactional database management systems (DBMS's) have special data integrity requirements that standard filesystems such as the Berkeley Fast Filesystem do not address. This paper briefly describes the requirements a transactional DBMS makes of a transaction-based filesystem, then goes on to describe the design and implementation of such a filesystem, referred to as a *block repository*<sup>1</sup>, which is part of the SQRL DBMS project.

The implementation of SQRL's *block repository* is different than most traditional filesystems in that it is purposely implemented in user-land using raw devices and threads. Its performance is more tunable to the needs of transaction processing than would be the case if it were integrated into the kernel.

## 1 Introduction

Transactional database management systems go to great lengths to never lose or corrupt data, even in cases of unexpected system failure. Algorithms that achieve atomic writes of data stored on disk are complex and can be very slow, depending on what support is available from the underlying filesystem. Traditional filesystems such as the Berkeley Fast Filesystem (FFS) guarantee atomic updates of filesystem metadata in order to avoid filesystem corruption caused by system failures, but no atomicity guarantees are made for file writes.<sup>2</sup> This means that in order to avoid possible file corruption, programmers of transaction-based applications have to do extra work to make atomic changes to files.

One of the simplest, though not most efficient, methods of implementing atomic writes on FFS is to use triple redundancy:

---

```
a = open("A", O_RDWR);
b = open("B", O_RDWR);
c = open("C", O_RDWR);
...
[Write changes to A.]
fsync(a);
[Write changes to B.]
fsync(b);
[Write changes to C.]
fsync(c);
...
```

---

If there is a system failure during writing, there are only three possible inconsistent states, all of which are fixable:

1.  $A \neq B = C$ . Fix:  $A \leftarrow B$ .
2.  $A \neq B \neq C$ . Fix:  $B \leftarrow A, C \leftarrow A$ .
3.  $A = B \neq C$ . Fix:  $C \leftarrow B$ .

---

A more practical implementation of atomic writes on FFS takes advantage of atomic metadata updates:

---

1. Definitions for emphasized words appear in the "Terms and Abbreviations" section toward the end of this paper.
2. FFS does not implement atomic file updates because the associated overhead is unacceptable for general use, since most applications do not need atomicity guarantees when writing to files.

```

/* "A" contains valid data. */
...
a_new = open("A_new", O_RDWR | O_CREAT,
0644);
[Write data to A_new.]
fsync(a_new);
rename("A_new", "A");

```

Recovery is as simple as deleting "A\_new", if it exists.

Transactional DBMS's must be able to atomically write data to persistent storage. As shown above, atomic writes can be achieved using standard filesystems. However, the performance of such schemes is far from ideal. The filesystem discussed in this paper is specially designed to meet the specific needs of a transactional DBMS. The filesystem discussed in this paper, hereafter referred to as a *block repository* or *BR*, has many architectural similarities to journaled filesystems. It differs though from most journaled filesystems in at least the following ways:

- Provides a simple block-oriented interface, as opposed to a file-oriented interface. The *BR* provides a mechanism for implementing data storage, but almost no policy.
- Implemented in user-land for improved performance and control. Rather than reading and writing data via system calls, a library is linked into the application. The *block repository* library assumes exclusive access of all storage resources that are allocated to it. This is undesirable for typical multi-process applications, but a useful simplification for single-process server applications.
- Data are stored on multiple devices, called backing stores. In this regard, the *BR* integrates and relies on some concepts normally found in volume management software.

## 2 Backing Store Creation

A *block repository* consists of four or more backing stores. A backing store is a file or raw device that consists of a header and data space, as

shown in Figure 1. The backing store header is triple-redundant so that atomic header updates can be guaranteed. The three copies of backing store header data are striped so that the first portion of each copy can be compared in order to detect and repair data corruption before reading in the remainder of the three copies of the header data. If the three copies of header data were not striped, then the first copy's notion of the header size would have to be trusted, which is not safe.

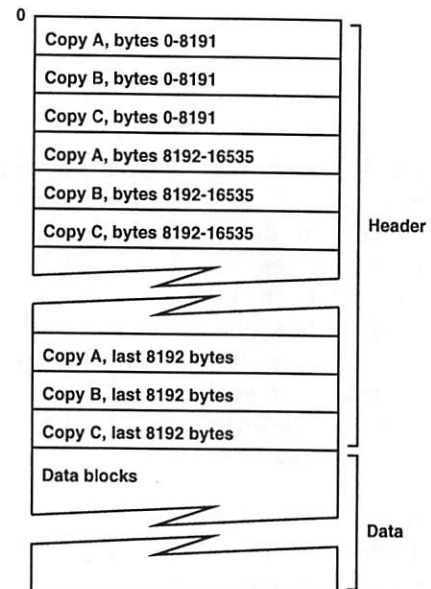


Figure 1: Backing Store Structure

The data stored in each copy of the backing store header is shown in Figure 2.

## 3 Block Repository Creation

Figure 3 shows the *block repository* structure. Before a *BR* can be brought online, at least a portion of each of the four logical sections of the *BR* must be backed by one or more backing stores.

Each backing store in the *BR* has a copy of the backing store list. When a change is made to the backing store list, the change is synchronously written to each backing store header, in the order that the backing stores are listed. Care is taken to write backing store header changes in this order in so as to assure the ability to recover from backing store headers that disagree

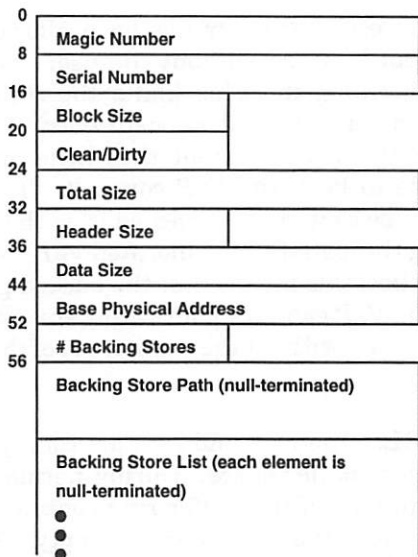


Figure 2: Backing Store Header Structure

with each other. A serial number is included in the header to aid the detection of differences between backing store headers.

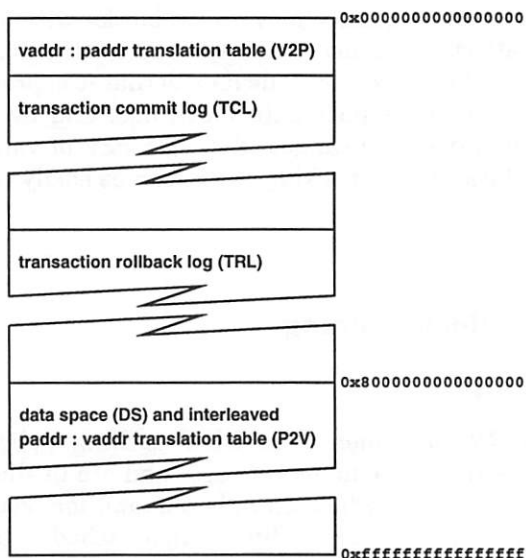


Figure 3: Block Repository Structure

## 4 Block Repository Startup and Recovery

As with data modifications that are made during normal *BR* operation, care must be taken during startup and recovery to never write data in such a way that could irreversibly corrupt the *BR*. The general startup and recovery algorithm is as follows:

1. Open a backing store, *S*, that is part of the *BR*.
2. While *S* is not the first backing store in *S*'s backing store list:
  - (a) Open the backing store, *S'*, that is first in *S*'s backing store list.
  - (b) Make *S'* the new *S*.
3. Repair *S* if necessary.
4. Repair and roll forward all backing stores in *S*'s list of backing stores to contain the same header data, if necessary.
5. Map all backing stores.
6. Roll forward the transaction commit log (*TCL*), if necessary.

## 5 Block Repository Operation

The *block repository* is designed to be able to stay online for long periods of time. This requires that all normal operations on the *BR* must allow uninterrupted availability of data. Below are brief descriptions of some common online *BR* operations.

### 5.1 Backing Store Operations

As mentioned earlier, a *BR* must be backed by at least four backing stores for it to be brought online. The *BR* design allows online insertion and deletion of backing stores, such that the *BR* can grow and shrink dramatically in size without ever having to be taken offline for maintenance or reconfiguration. Backings can overlap,

but there is no performance advantage or gain in resiliency to hardware failures in doing so. The sole reason for allowing overlapping backings is to make it possible to seamlessly move data from one device to another. Possible reasons for moving data from one device to another include:

- Consolidation of multiple small backings into one larger backing.
- Migration to a different storage technology.

Figure 4 shows an in-progress example of consolidating multiple backings (*A*, *B*, and *C*) into one backing (*D*). The algorithm for adding a new backing store is as follows:

1. Insert a backing store, *S*, into the *BR*, but mark *S* as invalid.
2. Begin writing all data writes that are in the range to be backed by *S* to *S*, in addition to any other backing stores that back any particular block.
3. Make a single sweep through the entire range of blocks to be backed by *S*, read the values of the blocks, and write them to *S*.
4. Mark *S* as valid.
5. Append *S* to the backing store list in each backing store header.

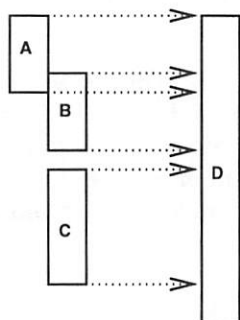


Figure 4: Consolidation of Multiple Backings

A backing store, *S*, can only be removed if there are one or more backing stores that also back the same blocks as *S* does, or if the blocks backed by *S* are not allocated, or a combination of both. The definition of “allocated” varies, depending on the logical section of the *BR*:

**V2P:** Once a *V2P* entry has been allocated, it cannot be freed without the user explicitly deallocating it. This limits the *BR*’s ability to compact and relocate entries in the *V2P* to such an extent that little effort is made to keep the *V2P* compact. The *V2P* must be contiguously backed from the beginning through the last allocated *V2P* entry. In practice, this means that the backed portion of the *V2P* can grow, but rarely shrinks substantially without the explicit aid of the user.

**TCL, TRL:** The *TCL* and *TRL* are conceptually rings of buffer space. During normal operation, part of the buffer ring contains valid data, and the other part is empty. At any point in time, the portion that contains valid data is considered to be allocated. A backing store, *S*, can only be removed from the *TCL* or *TRL* when no portion of *S* is the sole backing of an allocated region.

**DS/P2V:** The *DS* and *P2V* are subdivided into *extent groups*, which are discussed in more detail later. An *extent group* is the unit of allocation from the perspective of backing stores. If any data block within an *extent group* is allocated, the entire *extent group* must be backed. The *P2V* makes it possible to efficiently move data blocks without affecting the user’s ability to access the data blocks via *vaddr*. This means that it is practical to compact data block use, and even empty entire ranges of the *DS/P2V* of valid data, so that backing stores can be easily removed.

## 5.2 Block Caching

The *BR* implements an LRU caching policy. Blocks that have no lockers age, and are flushed from the cache when there is demand for block cache space due to reading of non-cached data. Naturally, this implies the constraint that there must be a large enough block cache to accommodate all concurrently locked blocks. In practice, for a reasonably sized block cache, this limitation is only an issue if faulty programming causes an accumulation of stale locks.

s	t	d	q	r	w	x	
✓		✓	✓	✓	✓	✓	s
	q		✓	✓	✓	✓	t
		✓	✓	✓	✓	✓	d
			✓	✓			q
				✓	✓		r
							w
							x

Figure 5: Block Lock Compatibility Matrix

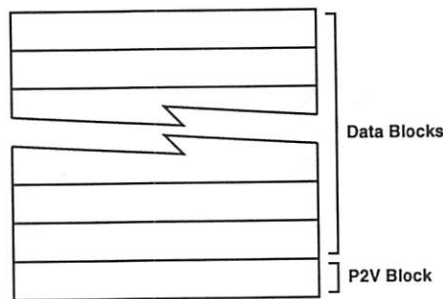


Figure 6: An Extent Group

### 5.3 Block Locking

Figure 5 shows the seven types of locks on data blocks, and their compatibility with each other. Following is a short description of each type of lock:

- s, t:** Non-serialized (s) and serialized (t) place holder locks. In order to obtain one of the other lock types, a user must first obtain an s or t lock.
- d:** Potential deletion lock. This is an advisory lock that supports a deletion algorithm for a form of B-trees.<sup>3</sup>
- q:** Non-exclusive read lock. This form of read lock is compatible with w locks, and should only be used when the reader can tolerate dirty reads of any sort that may be introduced by writers that hold w locks. The interactions between q locks and w locks offer a mechanism for allowing dirty reads, but no policy for how to deal with the effects of dirty reads.
- r:** Non-exclusive read lock.
- w:** Write lock that allows simultaneous q locks.
- x:** Exclusive write lock.

### 5.4 Data Block Management

Externally, data blocks are always accessed via *vaddr*. Internally, the V2P maps *vaddr*'s to *paddr*'s, and the P2V maps *paddr*'s to *vaddr*'s. Data blocks reside in the DS. The DS and P2V are interspersed such that one P2V block and a

number of DS blocks are grouped into an *extent group*, as shown in Figure 6.

Extents provide a mechanism for allocating multiple data blocks that are physically near each other. This generally improves the locality of data access, assuming that data blocks that are allocated in groups tend to be accessed in groups.

The *block repository*'s support for extents is in most ways simple. Extents always consist of a number of blocks that is a power of two, so that extents can be split and collated in logarithmic time. Once an extent is allocated, there is no longer a deterministic way to know the boundaries of the extent. In other words, it begins to be treated as some number of data blocks that have no explicit link to each other. This means that blocks that were allocated as an extent can be deallocated one at a time, and the worst thing that will happen is some extent fragmentation. Even this fragmentation is of limited concern though, since data blocks can be physically moved without external effects. Thus, by compacting allocated data blocks during light system load, extent fragmentation can be kept to a minimum.

Figure 7 shows a fully collated *extent group*. Since each *extent group* is followed by one P2V block, *extent groups* always consist of  $(2^n - 1)$  data blocks, where  $(6 \leq n \leq 16)$ . Figure 8 shows what a partially allocated *extent group* could look like. Note that neither of these figures distinguishes between allocated and free extents, though there is bookkeeping information in the P2V block that keeps track of allocation, so that extents can be

3. The original motivation for the block repository discussed in this paper is to support research on a highly concurrent B-tree algorithm.

collated with their neighbors as they are freed.

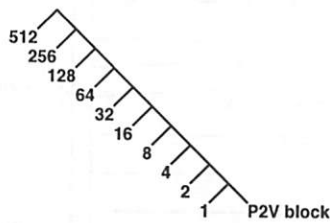


Figure 7: Fully Collated Extent Tree

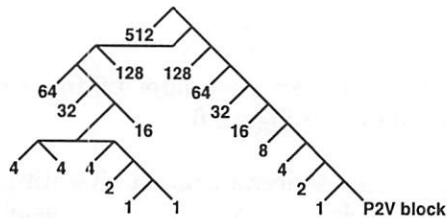


Figure 8: Partially Allocated Extent Tree

## 5.5 Transaction Commit Log Processing

All writes to the *V2P*, *DS*, and *P2V* sections of the *BR* are first recorded in the transaction commit log (*TCL*). At some later time, the log data are consumed by another thread of execution, here referred to as the cleaner. The cleaner reads the tail end of the *TCL*, and writes blocks to their permanent locations on disk. After each group of blocks in the *TCL* is written out, the cleaner marks the group as invalid, so that in the case of crash recovery, there is no need to process portions of the *TCL* that have already been taken care of.

Due to the interactions between the block cache and the *TCL*, there is a significant performance advantage to having a large amount of valid data in the *TCL* during normal operation. Suppose that a particular data block, *D*, is being written to quite regularly. Over time, the *TCL* will have recorded many different versions of *D*. When the cleaner comes across a block in the *TCL* that records a version of *D*, it first looks to see if *D* resides in the block cache. If so, the most recent version of *D* is written to its permanent location. If not, the cleaner can correctly assume that when *D* was flushed from the block cache, it was first written to the *TCL*, then written to its

permanent location on disk. This means that the cleaner, along with how the block cache flushes aged blocks, is in many cases able to write *D* to its permanent location only once, even if *D* was modified many times. Naturally, after a system fault, the entire *TCL* must be processed before the *BR* can be brought back online.

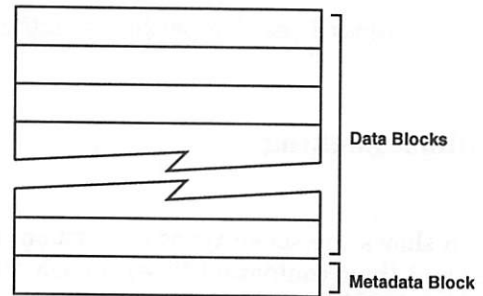


Figure 9: Structure of Transaction Commit and Rollback Logs

Each block of a log transaction has a metadata record associated with it, as shown in Figure 10. A log transaction consists of one or more data blocks. If a log transaction consists of four blocks, the transaction sequence numbers count from four down to one. The transaction sequence numbers also serve the purpose of “flip-flops”. If the “flip” and the “flop” are different, this indicates there was an incomplete write as a failure occurred. The flip-flops are necessary since the log metadata are not redundant, and in the case of log writes, redundancy would be more expensive from a performance perspective than the flip-flops.

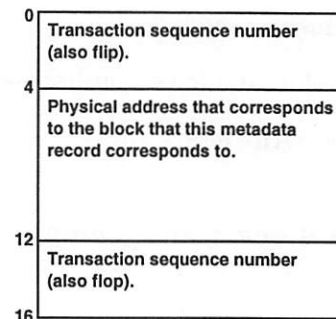


Figure 10: Log Metadata Record

## 6 Backup and Restore

The *block repository* supports both full and incremental online backup. A full backup is made using the following algorithm:

1. Note the current head of the *TRL*,  $H = \text{head}(\text{TRL})$ .
2. Do a block-wise copy to backup of the *V2P*.
3. Do a block-wise copy of all valid *DS* blocks, in such a way that during restore, each block can be associated with its *vaddr*.
4. Note the current head of the *TRL*,  $H' = \text{head}(\text{TRL}')$ .
5. Do a block-wise copy of all *TRL* blocks between  $H$  and  $H'$ .

An incremental backup can be made by copying the portion of the *TCL* written since the last full backup was made.

*BR* restoration is accomplished by restoring a full backup, then, in chronological order, restoring any incremental backups that were made.

## 7 Summary

Traditional filesystems do not provide the combination of atomic data writes and a simple block-oriented interface that transactional DBMS's require. By integrating parts of existing filesystem technologies, a simple streamlined data storage mechanism can be created that meets the needs of transactional systems without the complexity and non-portability of explicit operating system kernel support.

Due to the *block repository*'s user-land nature and its use of multiple devices for storage, performance can be highly tuned to the needs of the individual application.

## Terms and Abbreviations

**BR:** Block repository. For the purposes of discussion in this paper, a block repository is similar in many ways to a conventional filesystem, but operations are on data blocks with a flat 64 bit numerical namespace rather than on files in a hierarchical textual namespace.

**vaddr:** Virtual address. From the user's perspective, all data blocks are accessed by specifying a *vaddr*.

**paddr:** Physical address. Internally, the *BR* consists of a 64 bit "physical" address space.

**V2P:** *vaddr* to *paddr* translation table. The *V2P* makes it possible to find the *paddr* of any data block, given the *vaddr*.

**backing store:** Encapsulation of a device or file that provides non-volatile storage for a portion of the block repository's *paddr* space.

**TCL:** Transaction commit log. All data writes are first written to the *TCL* in such a way that once an entire transaction has been written to the *TCL*, there is enough information to be able to sometime later write the data to their permanent locations, even if there is a crash in between.

**TRL:** Transaction rollback log. Pristine copies of all modified data blocks are written to the *TRL* before modified data are written to the *TCL*. This provides a reliable method for restoring the state of the *BR* to a previous state, as well as supporting online snapshot backups.

**DS:** Data space. Data blocks are stored here.

**P2V:** *paddr* to *vaddr* translation table. The *P2V* is used for various internal algorithms to move data blocks around without causing any externally visible changes. The *P2V* also contains part of the information that is used to implement extents.

**extent group:** An extent group consists of a *P2V* block and a set of *DS* blocks. The *P2V* block stores metadata that correspond to the *DS* blocks in the extent group.

## Availability

The block repository described in this paper is part of SQRL, which is an ongoing project sponsored by the Hungry Programmers (<http://www.hungry.com>) to create a free SQL-92 DBMS. Information and current source code for SQRL can be found at <http://www.sqrl.org/sqrl>.

All software that is part of SQRL is released under a very agreeable BSD-like license.

## References

- [Bernstein] Phillip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley Publishing Company, Inc. (1987).
- [Elmasri] Ramez Elmasri and Shamkant B. Navathe, *Fundamentals of Database Systems, Second Edition*, Addison-Wesley Publishing Company, Inc. (1994).
- [Folk] Michael J. Folk and Bill Zoellick, *File Structures, Second Edition*, Addison-Wesley Publishing Company, Inc. (1992).
- [Gray] Jim Gray and Andreas Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Publishers, Inc. (1993).
- [McKusick] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*, Addison-Wesley Publishing Company, Inc. (1996).

# Standalone Device Drivers in Linux

Theodore Ts'o

April 28, 1999

## 1 Introduction

Traditionally, Unix device drivers have been developed and distributed inside the kernel. There are a number of good reasons why this method is the predominant way most device drivers are distributed. First of all, it simplifies the packaging and distribution issues for the device driver. Secondly, it makes it easier to make changes to the interfaces between the kernel and the device drivers.

However, this traditional scheme has a number of major disadvantages. First of all, it means that each version of the device driver is linked to a specific version of the kernel. So if a device is only supported in a development release, a customer who might otherwise want to use a stable, production release might be forced to use a bleeding-edge system. Alternatively, there may be bugs present in the device driver as included in the stable release of the kernel, but which are fixed in the development kernel.

Moreover, including device drivers with the kernel is not scalable in the long term. If Linux-like<sup>1</sup> systems are ever to be able to support as many devices as various OS's from Redmond, Washington, hardware manufacturers will have to be able to support and distribute drivers separately from the main kernel distribution. Currently, the size of the total Linux kernel source base has been doubling (roughly) every 18 months. Almost all of this growth has been the result of new device drivers. If Linux is to be successful at Linus Torvald's goal of Total World Domination, it will be essential that we remove any limits to growth, and an exponential growth in the size of the Linux kernel is an obviously a long-term growth limitation.

## 2 Existing systems

Currently, there are a number of device drivers which are currently distributed outside of the Linux kernel: The Comtrol Rocketport and VS-1000 drivers, written by the author, and David Hind's PCMCIA drivers.

The Comtrol Rocketport driver is notable in that a single source base can support three stable Linux releases (1.2, 2.0, 2.2) as well as a number of the intervening development releases. It was necessary to support a wide range of Linux releases because the a number of customers of Comtrol Rocketport board were Internet Service Providers (ISP's) who were using the intelligent serial board to provide dialup access of various kinds (PPP, UUCP, login, etc.). For these customers, stability is paramount, and often this causes them to defer upgrading their systems for a long time.

David Hind's PCMCIA drivers present a very good example of some of the advantages of stand-alone device drivers. Hardware manufacturers are constantly introducing new PCMCIA devices, and often these new require making minor tweaks to the PCMCIA package. These frequency of such changes often exceeded

---

<sup>1</sup>The term "Unix(tm)-like" is frowned upon by the Open Group copyright police :-)

the release frequency of the stable kernel series. If the PCMCIA driver had been integrated into the mainline kernel code, then users of the stable kernel series might be forced to use the development kernels if they needed to use a newer PCMCIA device. The alternative, trying to merge changes back and forth between the stable and kernel series, and forcing frequent releases of the stable kernel, is almost equally unpalatable. The solution, then, of distributing the PCMCIA drivers separately from the main body of the Linux kernel sources, was almost certainly the best way of solving this conundrum.

### **3 Techniques for writing stand-alone drivers in Linux**

Unfortunately, the structure of Linux kernel is such that stand-alone drivers is not easy. This is perhaps the reason why there are so relatively few drivers which are distributed in this way. Unfortunately, the various interfaces used by device drivers are not fixed, and change over time. (This is not unique to Linux; many Unix(tm) systems have similar issues.) Hence, the techniques used by existing standalone device drivers utilize a certain amount of brute-force. However, it is surprising that despite the a certain amount of elegance, it is easier than it first seems to make achieve this goal.

#### **3.1 Using Kernel Modules**

The first way device drivers were distributed separately from the kernel were to simply distribute kernel patches which dropped the driver into the kernel sources. The system administrator would then configure the kernel to use the new driver, and recompile the kernel. While this provides the most easy and simplicity for the driver author, it is less than convenient for the users of the driver who have to compile, install, and use the driver.

Loadable kernel modules were originally introduced to solve a variety of problems. They allow a developer to try out their driver without having to reboot a new kernel, thus drastically shorting the edit, compile, and test cycle. They also allow a single common set of kernel and modules to be used across a wide variety of machines, while avoiding the expense of statically compiling a large number of drivers into a a single, large, generic kernel. This is very important for Linux distributions, since many of their customers may not be comfortable with configuring and compiling their own kernel.

Although most kernel modules today are distributed as part the Linux kernel, kernel modules turn out to be an extremely useful distribution mechanism for standalone device drivers as well. All of the advantages for modules distributed with the kernel sources apply doubly so for modules distributed separate from the kernel. Perhaps most importantly, in many cases the user doesn't even need to have the kernel sources installed on their system; a set of kernel include files which correspond to the currently installed kernel is sufficient.<sup>2</sup>

#### **3.2 Building modules outside of the kernel tree**

While writing a device driver which is to be included in the standard kernel source tree, the author can rely on the build infrastructure provided by the kernel. The author, however, can no longer depend on this infrastructure when distributing her device driver as in a standalone package.

Fortunately, the Linux kernel does not require an overly complex build system in most cases. (More on the exceptions in a moment). For the simplest modules, all that is required is that the following C preprocessor

---

<sup>2</sup>Unfortunately, some more primitive distributions have distributed a set of kernel include files which do match with the kernel which they are distributing. Certain versions of Slackware distribution in particular have suffered from this problem.

flags be defined by the Makefile: `__KERNEL__` and `LINUX`. If the kernel was configured to be built using SMP, the Makefile must also define preprocessor flag `__SMP__`. In addition, the C compiler needs to be told where to find the kernel include files, so a compiler directive such as `-I/usr/src/linux/include` is generally required.

In some cases, it may be desirable to write the kernel module so that it can be used either inside the kernel source tree, or separately in a standalone distribution. In these cases, the driver Makefile may contain a definition of some flag such as `-DLOCAL_HEADERS` which is not defined in the standard kernel build environment, so that the driver code can distinguish between the standalone compilation environment and the kernel compilation environment. In this example, the flag is used so that the driver includes the local header files (which may have required changes or improvements), instead of the header files found in the standard kernel header files.<sup>3</sup>

### 3.2.1 Synchronizing kernel versions

A very common cause of user confusion and complaints is caused by mismatches between the version of the kernel header files, and the kernel actually installed and running on the system. This can occur due to a number of reasons. Some distributions erroneously include inconsistent kernel and kernel header files. Other users may have installed a set of kernel sources, but haven't yet installed the new kernel yet.

In most cases, the symptom reported by users suffering from this header file/kernel synchronization problem is simply that the `insmod` utility refuses to load the newly built device driver. However, sometimes the kernel module will load, but then fail in some mysterious way. Mercifully, these cases were fairly rare, because they are extremely difficult to debug.

In order to avoid these problems, it is a very good idea to have the makefile determine the kernel version of the header files, as well as the kernel version of the currently running kernel, and issue a warning if these two versions are different. Under Linux, the kernel version string should also include critical configuration information about whether the kernel is an SMP kernel (and thus requires the `-D__SMP__` flag when compiling the module object files) and whether `CONFIG_MODULEVERSIONS` option has been enabled.<sup>4</sup>

The makefile also should maintain a file which contains the kernel version of the header files. This file serves two purposes. It is used by the installation utility (see section 3.3). In addition, the makefile can use this file to determine when the version of the header files have changed, and force all of the object files to be rebuilt if necessary, thus avoiding another common user-reported problem.

### 3.2.2 Exporting symbols from a module

The job of a kernel module's Makefile becomes much more complicated if the module exports functional interfaces which are used by other modules. This is due to a requirement (if the `CONFIG_MODULEVERSIONS` option is enabled) to include a 32-bit checksum in the symbol names of functions which are exported for use by modules, no matter whether those functions are defined by the kernel or by other modules. The 32-bit checksum is calculated over the function signature<sup>5</sup> including all of the definition of the types used by the function signature. This will prevent a previously compiled kernel module from linking with a new kernel

<sup>3</sup>This can also be accomplished by adding a `-I.` directive ahead of the `-I/usr/src/linux/include` directive, so that local header files are used in preference to the standard kernel header files. This approach has the advantage of minimizing the number of `#ifdef`'s in the code. However, it means that the header files must be laid out in the same include file hierarchy as was used inside the kernel.

<sup>4</sup>See section 3.2.2 for a discussion why this is necessary.

<sup>5</sup>That is, the types of all of its arguments as well as the type returned by the function.

if the function prototype changes, or any of the type definitions change, the changed checksum will cause the linker in the `insmod` utility to fail, since the checksum is part of the symbol name.

If the module is only using kernel-provided functions, the `modversions` scheme is very easy to use; a series of C preprocessor macros in the kernel header files automatically redefine references to functions such as `kmalloc` with a symbol name such as `kmalloc_R93d4cfe6`. However, if a module also needs to export functions for use by other modules, the makefile must generate and modify the kernel header files, and this turns out to be rather tricky.

The method of specifying which symbols need to be exported changed between Linux 2.0 and Linux 2.2. Hence, modules that need to support both older and newer versions of Linux will need to have something like the following in their source:

```
#if (LINUX_VERSION_CODE > 0x20100)
EXPORT_SYMBOL(register_serial);
EXPORT_SYMBOL(unregister_serial);
#else
static struct symbol_table serial_syms = {
#include <linux/symtab_begin.h>
    X(register_serial),
    X(unregister_serial),
#include <linux/symtab_end.h>
};
#endif
```

Similarly, the method for invoking `genksyms`, the program which examines a source file and its header files to generate symbol checksum, has also changed over between Linux 2.0 and 2.2. In addition, if the module is being built for an SMP kernel, an appropriate argument to `GENKSYMS` must be passed by the Makefile to indicate this in the symbol checksums. All of this requires that the Makefile be cognizant of the kernel version of its compilation environment, which is another good reason why the Makefile should store this information in a conveniently accessible file:

```
serial.ver: serial.c $(DEP_HEADERS)
    @echo "Generating serial.ver..." ; \
    if test -s .smpflag ; then smp_prefix="-p smp_"; \
    else smp_prefix=; fi; \
    kver=`sed -e 's/-SMP//' -e 's/-MOD//' .kver` ; \
    $(CC) $(GENKSYM_FLAGS) -E serial.c > serial.tmp ; \
    echo $(CC) $(GENKSYM_FLAGS) -E serial.c \> serial.tmp ; \
    case $$kver in \
    2.0.*) echo $(GENKSYMS) . < serial.tmp ; \
           $(GENKSYMS) . < serial.tmp ;; \
    2.[12].*) echo $(GENKSYMS) $$smp_prefix -k $$kver \< serial.tmp ; \
              $(GENKSYMS) $$smp_prefix -k $$kver < serial.tmp \
              > serial.ver ;; \
    *)      echo "Unsupported kernel version $$kver" ;; \
    esac ; rm -f serial.tmp ; echo rm -f serial.tmp ; \
    mkdir -p linux/modules ; cp serial.ver linux/modules ; \
```

One final subtlety which must be considered is if `.ver` file containing the symbol versions is not mentioned in the kernel header's `modversions.h` file, the Makefile must generate its own copy of the this file which

has been modified to include this `.ver` file, and place it in an appropriate location inside the module build tree so that it is used instead of the system `modversions.h` header file.

### 3.3 Installing modularized device drivers

As many a software developer knows, it is not enough to write wonderful programs; in order to be used, the user has to be able to install it! Unfortunately, it is all-too-common in the Linux community (and in the Unix community in general) for developers to pay little attention to the installation process. This is perhaps partially responsible for the reputation which Linux and Linux-like systems have of being user-hostile.

What needs to be done during installation? Obviously, the object file containing the modularized device driver needs to be installed in its proper location in the `/lib/modules` hierarchy. The installation should install the module both in a kernel-version dependent location, such as `/lib/modules/2.2.1/misc/serial.o` and in a version independent location, such as `/lib/modules/serial.o`. The kernel-version dependent location *must* be based on the kernel version for which the driver was compiled, and not the kernel version currently running on the machine at the time of the installation. (However, it is a good idea for the installation program to check to see if these two kernel versions are different, and print a warning message if they do not.

Secondly, if the driver is loaded at boot-time or if it requires some kind of special initialization, the installation program will need to install a `/etc/rc.d` script. There are several reasons why a device driver might need a boot-time script. For example, the driver might need to be loaded at all times, instead of being dynamically loaded by `kerneld` or the `kmod` thread. Or perhaps the drivers require special run-time configuration.<sup>6</sup>

Unfortunately, the exact location for installing the `/etc/rc.d` file varies from distribution to distribution. This means the shell script must detect the `/etc/rc.d` convention currently in use on the system, and adapt its installation accordingly.

Finally, if the module exports any functions (see section 3.2.2) the installation script will need to modify the kernel header files so that other modules will be able to successfully use the driver's exported interfaces. (Note however that these kernel header files may be modified and revert back to the original contents if a `make depend` is executed in the kernel source tree.

Because of the potential complexity of the installation script, while it can be done as a Makefile target (i.e., what happens when you type `make install`), most of the time the steps required to install a device driver are best stored in a shell script.

### 3.4 Kernel version compatibility

The Linux kernel interfaces have changed over time. Although a device driver developer sometimes suspects the changes are made only to make his life more difficult, in reality the changes are needed to make the kernel code be more general or more performant.

A good example of this is the changes which were made to the interfaces used by kernel routines to read or write user memory which were made during the 2.1 development cycle for the i386 architecture. Instead of manually calling the `verify_area()` routine which would check the page tables to see whether or not the memory access was allowed (or whether a page needed to be copied or faulted into memory), the new routines used the i386 memory management routines to notice if an access violation or page fault occurred, and the fault handler looked up the faulting PC in a table to determine which exception handler should

<sup>6</sup>In many circumstances, dynamically loaded drivers can be configured automatically by `modprobe` when they are loaded by making appropriate changes to the `/etc/conf.modules` file.

be invoked to recover from the memory fault. Unfortunately, the kernel API used in Linux 2.0 was not general enough to allow for this particular optimization — hence, the interface was changed during the 2.1 development cycle.

When functional kernel interfaces changes, the best way for an author to adapt her code is to change the code to use the new interface, and then to provide backwards compatibility routines which implement the new interface in terms of the older interfaces. For example, the following routine can be found in the stand-alone serial driver:

```
#if (LINUX_VERSION_CODE < 131336)
static int copy_from_user(void *to, const void *from_user, unsigned long len)
{
    int      error;

    error = verify_area(VERIFY_READ, from_user, len);
    if (error)
        return len;
    memcpy_fromfs(to, from_user, len);
    return 0;
}
#else
#include <asm/uaccess.h>
#endif
```

In the above example, `LINUX_VERSION_CODE` macro is used to determine the version of the kernel for which the driver is being compiled. It is set by the kernel headers using the following formula:  $version * 65536 + patchlevel * 256 + sublevel$ . Hence, the Linux kernel version 2.1.8 corresponds to a `LINUX_VERSION_CODE` of  $2 * 65536 + 1 * 256 + 8$ , or 131336.

In other cases, a the definition of a structure may change. For example, a new structure element might get added in the middle of the structure. In other cases, the structure elements might get reordered for performance reasons. A good way of insulating a program from such changes is to avoid the use of initializers, i.e.:

```
static struct console sercons = {
    "ttyS",
    serial_console_write,
    NULL,
    serial_console_device,
    .
    .
    .
};
```

Instead, it is safer to initialize the structure explicitly, like this:

```
static struct console sercons;

memset(&sercons, 0, sizeof(sercons));
strcpy(sercons.name, "ttyS");
```

```
sercons.write = serial_console_write;
sercons.device = serial_console_device;
```

This approach is not without its dangers; for example, new structure elements may have to be properly initialized in or the kernel abstraction will not work correctly. So while this technique does not eliminate the need for testing the driver as kernel interfaces change, it does minimize the number of version specific changes which need to be made to the driver source.

## 4 Future work

Many of the techniques described in this paper will no doubt strike the reader as being extremely hackish and kludgy. There is unfortunately more than a grain of truth in such an observation. However, the techniques described in this do work, and have been successfully used in more than a few device drivers. Authors of device drivers are urged to try to try distributing works in standalone form despite the primitive nature of the support currently available for such distributions, because of the many advantages which we have discussed in this paper.

That being said, there are many ways in which Linux could be improved in order to make stand-alone device drivers distribution much cleaner, simpler, and elegant. Let us discuss some of the possibilities, with the hopes that future versions of Linux will incorporate at least some of these improvements.

### 4.1 Standards for installing rc.d scripts

It is extremely unfortunate that the various Linux distributions have not been able to standardize on a single standard for `/etc/rc.d` init files.<sup>7</sup> This was mainly due to an accident of history, in that the various distributions did not coordinate before adopting their `/etc/rc.d` conventions, and now do not feel that they can easily migrate to a single standard without compromising their backwards compatibility with older versions of their distribution.

The Linux Standard Base (LSB) effort has been working on this and other problems which have made life difficult for Independent Software Vendors (ISV's) who wish to distribute software which is compatible with all, or at least most, Linux distributions. The solution which has been proposed for solving this problem is a configurable tool which can be called by installation scripts who need to have their `/etc/rc` script installed at boot time. This tool would read structured comments placed at the beginning of the `/etc/rc` script to determine where in the logical boot and shutdown sequences the script needed to be called, and based on the conventions of that particular distribution, the tool would install the script and the necessary symbolic links in the proper places. While not as satisfying as simply being able to declare a standard for `/etc/rc` files, it does prove the adage that any computer science problem can be solved by adding an extra layer of indirection.

### 4.2 Binary Compatibility

All of the techniques described in this paper assumed that part of the installation process for the driver involved compiling the driver from source for a particular machine. Unfortunately, it is not at all clear whether or not this will suffice for the vast majority of the non-technically inclined user base which Linux

---

<sup>7</sup>Even worse is the Slackware distribution, which is extremely primitive. It doesn't even support the System V-style init files at all. This means that it is extremely difficult to programmatically install a script to be run at boot-time. System administrators usually have to manually edit their `/etc/rc` files by hand.

will need to start attacking if it is to continue its current exponential growth rate. Even if the source to the drivers are available, the thought of needing to compile a kernel extension may scare away more naive users. Furthermore, their system might not even have the C compiler installed!

Currently, interfaces change often enough, particularly at the binary level, that it is impractical to ship binary modules. Even during the course of a stable release, the core Linux developers have refused to “tie their hands” by making commitments that the ABI exposed by the kernel module would remain unchanging. Their concern is that some critical bug fixes might require making changes to the structures which would affect the ABI.

One way of addressing these two concerns is to add an adaptor layer which provides this fixed ABI. This adaptor layer might change over time to take advantage of new features in the kernel, but it would be very careful to maintain backwards compatibility at the ABI layer. The disadvantage of using such an adaptor layer is an inevitable performance loss. However, with sufficient tuning this performance loss might be made small enough to be insignificant, and in any case many might view a small performance loss to be an acceptable tradeoff in order to provide a standardized ABI for device drivers which are distributed in binary form.

### 4.3 Cross OS compatibility

Taking this the idea of compatibility one step further, there are those who believe it would be useful to provide an adaptor layer which would allow a single implementation of a device driver to serve multiple operating systems; further, that this be doable both on a binary and source level. There is currently a industry consortium of vendors who are working on just such a project, which has been named UDI, for “Uniform Device Interface.”

SCO Unix is a major backer of this initiative, and they are hosting the UDI web page, which can be found at <http://www.sco.com/UDI>. Furthermore, SCO has offered to write a proof-of-concept adaptor layer which would allow Linux to use UDI device drivers. Most developers in the Linux community have viewed this initiative with ambivalence. On the one hand, if successful it would greatly increase the number of device drivers supporting exotic hardware devices. On the other hand, there are some genuine performance issues which are raised by such a scheme. Can a device driver which is written to be general across multiple different types of OS and OS programming styles really be efficient? Only time will tell; the results of the SCO UDI effort should be most interesting.

## 5 Conclusion

In this paper, we have discussed the motivations for distributing device drivers separately from the kernel, and examined some techniques used by the few device drivers which are currently being distributed separately from the mainline Linux kernel. Finally, we have reviewed some strategies for making it easier for device drivers to be packaged and distributed in this way. Although much work remains to be done in order to make standalone device driver development easier, I hope that this paper encourages developers to consider distributing device drivers outside of the kernel, either instead of or in addition to including a version of the driver in the latest development kernel.

In the future, if Linux continues to grow in size and importance, hardware manufacturers who today only feel obligated to supply Windows driver may decide that they too wish to write and distribute a Linux driver to increase the number customers willing to buy their product. It is my hope that at that point, Linux will have advanced sufficiently that a less technically inclined user can simply pull the new scanner, digital camera out of the box, plug it into her computer, insert a device driver diskette and be using the new hardware moments

later. I believe that the use of stand-alone device drivers will make this type of scenarios much more likely to occur.

## A A Sample Makefile

The following makefile is taken from the author's stand-alone serial driver distribution:

```
LINUX_SRC=      /usr/src/linux

INCL_FLAGS=     -I. -I$(LINUX_SRC)/include
DEF_FLAGS=      -D__KERNEL__ -DLINUX -DLOCAL_HEADERS -DEXPORT_SYMTAB
BASE_FLAGS=     -m486 -O2 -Wall $(INCL_FLAGS) $(DEF_FLAGS) `cat .smpflag`

CFLAGS=        $(BASE_FLAGS) `cat .modflag`
GENKSYM_FLAGS=  $(BASE_FLAGS) -D__GENKSYMS__

CC=            gcc
GENKSYMS=      genksyms

VERFILE=       $(LINUX_SRC)/include/linux/modules/serial.ver
MODVERFILE=    $(LINUX_SRC)/include/linux/modversions.h

DEP_HEADERS=    serial.h serialP.h version.h \
               $(LINUX_SRC)/include/linux/version.h

all: test_version maybe_modules serial.o

test_version:
    @ISMP=''; KSMP=''; cp /dev/null .smpflag; \
    if grep -q smp_ $(LINUX_SRC)/include/linux/modules/ksyms.ver ; then \
        ISMP='-SMP'; echo "-D__SMP__" > .smpflag ; \
    fi ; \
    if grep -q smp_ /proc/ksyms ; then \
        KSMP='-SMP'; \
    fi ; \
    IMOD=''; KMOD=''; cp /dev/null .modflag; \
    if grep -q "#define CONFIG_MODVERSIONS" \
        $(LINUX_SRC)/include/linux/autoconf.h ; then \
        IMOD='-MOD'; echo "-DMODVERSIONS" > .modflag; \
    fi ; \
    if grep -q "kfree" /proc/ksyms && \
        ! grep -q "kfree$$" /proc/ksyms ; then \
        KMOD='-MOD' ; \
    fi ; \
    IVER=`grep UTS_RELEASE $(LINUX_SRC)/include/linux/version.h | \
        awk '{print $$3}' | tr -d \"'$$IMOD$$ISMP; \
    KVER=`uname -r`$$KMOD$$KSMP; \
    if [ -f .kver ] ; then \
        OVER=`cat .kver`; \
```

```

        if [ $$OVER != $$IVER ]; then \
            echo "Removing previously built driver for Linux $$OVER"; \
            make clean; \
            echo " "; \
        fi ; \
    else \
        make clean ; \
    fi ; \
    echo $$IVER > .kver ; \
    echo Building serial driver for Linux $$IVER ; \
    echo " " ; \
    if [ $$IVER != $$KVER ]; then \
        echo "WARNING: The current kernel is actually version $$KVER." ; \
        echo " " ; \
    fi

maybe_modules:
    @if test -s .modflag ; then \
        make modversions.h serial.ver ; \
    fi

install:
    @LINUX_SRC=$(LINUX_SRC) sh ./setup.sh

clean:
    rm -f serial.o *~ .kver serial.ver modversions.h
    rm -rf linux

really-clean: clean
    rm -f *.o

serial.o: serial.c $(DEP_HEADERS)
    @echo $(CC) $(CFLAGS) -c serial.c -o serial.o ; \
    $(CC) $(CFLAGS) -c serial.c -o serial.o

serial.ver: serial.c $(DEP_HEADERS)
    @echo "Generating serial.ver..." ; \
    if test -s .smpflag ; then smp_prefix="-p smp_"; \
    else smp_prefix=; fi; \
    kver=`sed -e 's/-SMP//' -e 's/-MOD//' .kver` ; \
    $(CC) $(GENKSYM_FLAGS) -E serial.c > serial.tmp ; \
    echo $(CC) $(GENKSYM_FLAGS) -E serial.c \> serial.tmp; \
    case $$kver in \
        2.0.*) echo $(GENKSYMS) . < serial.tmp ; \
            $(GENKSYMS) . < serial.tmp ;; \
        2.[12].*) echo $(GENKSYMS) $$smp_prefix -k $$kver \< serial.tmp ; \
            $(GENKSYMS) $$smp_prefix -k $$kver < serial.tmp \
                > serial.ver ;; \
        *) echo "Unsupported kernel version $$kver" ;; \
    esac ; rm -f serial.tmp ; echo rm -f serial.tmp ; \

```

```

mkdir -p linux/modules ; cp serial.ver linux/modules ; \
modversions.h: $(MODVERFILE)
    @if ! grep -q /serial.ver $(MODVERFILE) ; then \
        sed -f modversions.sed < $(MODVERFILE) > modversions.tmp ; \
    else \
        cp $(MODVERFILE) modversions.tmp ; \
    fi ; \
if ! test -f modversions.h || \
    ! cmp -s modversions.h modversions.tmp ; then \
    echo "Generating modversions.h" ; \
    mv modversions.tmp modversions.h ; \
    mkdir -p linux ; cp modversions.h linux ; \
else rm -f modversions.tmp ; fi

```

## B A sample installation script

The following installation script, `setup.sh` is taken from the author's standalone serial driver distribution:

```

#!/bin/sh
#
# This script is responsible for installing the Serial driver
#

if [ `whoami` != root ]; then
    echo "You must be root to install the device driver!"
    exit 1
fi

# Install a copy of the driver in the version independent location
if [ ! -d /lib/modules ]; then
    mkdir -p /lib/modules
fi

if test -d /lib/modules/misc ; then
    cp serial.o /lib/modules/misc
    rm -f /lib/modules/serial.o
else
    cp serial.o /lib/modules
fi

# Install a copy of the driver in a kernel version specific location
if [ -f .kver ]; then
    KVER=`sed -e 's/-SMP//' -e 's/-MOD//' .kver`
    KVERSION=`cat .kver`
else
    KVER=`grep UTS_RELEASE /usr/include/linux/version.h | \

```

```

        awk '{print $3}' | tr -d \"`
KVERSION=$KVER
if grep -q "#define CONFIG_MODVERSIONS" \
        $LINUX_SRC/include/linux/autoconf.h ; then
        KVERSION=${KVERSION}-MOD
fi
if grep -q smp_ $LINUX_SRC/include/linux/modules/ksyms.ver ; then
        KVERSION=${KVERSION}-SMP
fi
fi

if grep -q smp_ /proc/ksyms ; then
        RSMP='-SMP'
fi
if grep -q "kfree" /proc/ksyms &&
        ! grep -q "kfree$$" /proc/ksyms ; then
        RMOD='-MOD'
fi
RVERSION=`uname -r`$RMOD$RSMP
echo " "
echo "Installing serial driver for Linux $KVERSION"
echo " "
if [ $KVERSION != $RVERSION ] ; then
        echo "WARNING: The current kernel is actually version $RVERSION."
        echo " "
fi

#
# Actually install the kernel module
#
if [ ! -d /lib/modules/$KVER/misc ] ; then
        mkdir -p /lib/modules/$KVER/misc
fi

cp serial.o /lib/modules/$KVER/misc

#
# Now that we're done, run depmod -a so that modprobe knows where to find
# stuff
#
depmod -a

#
# Since the serial driver exports symbols, update the serial.ver and
# modversions file.
#
if test -z "$LINUX_SRC" ; then
        echo "Defaulting linux sources to /usr/src/linux"
        LINUX_SRC=/usr/src/linux
fi

```

```

VERFILE=$LINUX_SRC/include/linux/modules/serial.ver
MODVERFILE=$LINUX_SRC/include/linux/modversions.h

if ! test -f $VERFILE || ! cmp -s serial.ver $VERFILE ; then
    echo "Updating $VERFILE"
    cp serial.ver $VERFILE
fi
if ! grep -q /serial.ver $MODVERFILE ; then \
    echo "Updating $MODVERFILE to include serial.ver"
    sed -f modversions.sed < $MODVERFILE > ${MODVERFILE}.new
    mv $MODVERFILE ${MODVERFILE}.old
    mv ${MODVERFILE}.new $MODVERFILE
fi

#
# OK, now we install the installation script
#
if [ -d /etc/rc.d/init.d ]; then
    # This is a RedHat or other system using a System V init scheme
    RC_DEST=/etc/rc.d/init.d/serial
    RC_DIR=/etc/rc.d
    RC_START=S83serial
    RC_STOP=K18serial
elif [ -d /etc/init.d ]; then
    # This is a Debian system

    RC_DEST=/etc/init.d/serial
    RC_DIR=/etc
    RC_START=S83serial
    RC_STOP=K20serial
else
    # This must be a Slackware or other non-SysV init system
    if [ ! -f /etc/rc.d/rc.serial ]; then
        echo "The initialization script will be installed in "
        echo "/etc/rc.d/rc.serial.  You will need to edit your "
        echo "/etc/rc files to run /etc/rc.d/rc.serial"

        if [ ! -d /etc/rc.d ]; then
            mkdir -p /etc/rc.d
        fi
    fi

    RC_DEST=/etc/rc.d/rc.serial
    RC_DIR=""
fi

#
# Determine the version numbers of the installed script (if any) and the
# rc.serial script which we have.

```

```

#
SRC_VERS=`grep "^# FILE_VERSION: " rc.serial | awk '{print $3}'`
DEST_VERS=0
if test -f $SRC_DEST ; then
    DEST_VERS=`grep "^# FILE_VERSION: " $SRC_DEST | awk '{print $3}'`
    if test -z "$DEST_VERS" ; then
        DEST_VERS=0
    fi
fi

#
# Only update the destination file if we have a newer version.
#
if test $SRC_VERS -gt $DEST_VERS ; then
    if test -f $SRC_DEST ; then
        echo "Updating $SRC_DEST"
        NEW_INSTALL=no
    else
        echo "Installing $SRC_DEST"
        NEW_INSTALL=yes
    fi
    cp rc.serial $SRC_DEST
fi

if test -n "$SRC_DIR" ; then
    rm -f $SRC_DIR/rc?.d/[SK]??rc.serial
    if test "$NEW_INSTALL" = "yes" ; then
        echo " "
        echo "You are using a system which uses the System V init"
        echo "scheme. The initialization script has been installed"
        echo "in $SRC_DEST and it has been automatically "
        echo "set up to be run when you reboot"

        ln -sf ../init.d/serial $SRC_DIR/rc2.d/${RC_START}
        ln -sf ../init.d/serial $SRC_DIR/rc3.d/${RC_START}
        ln -sf ../init.d/serial $SRC_DIR/rc5.d/${RC_START}
        ln -sf ../init.d/serial $SRC_DIR/rc0.d/${RC_STOP}
        ln -sf ../init.d/serial $SRC_DIR/rc1.d/${RC_STOP}
        ln -sf ../init.d/serial $SRC_DIR/rc6.d/${RC_STOP}
    fi

    if [ -f /etc/rc.d/rc.serial ] ; then
        echo "You have an old /etc/rc.d/rc.serial script"
        echo "After checking to make sure it is no longer being used"
        echo "you should delete it."
    fi
fi

```

# Design and Implementation of Firewire device driver on FreeBSD

Katsushi Kobayashi

*Communication Research Laboratory, JAPAN*

ikob@koganei.wide.ad.jp

## Abstract

A Firewire device driver has been implemented on FreeBSD system. The driver provides IP network stack, native socket system interface, and stream device interface such as a DV video. The device driver shows enough performance on the IP over Firewire environment at 30Mbps. Also, DV video communication application using IP has been developed with the device driver and it enables DV quality communication between US and Japan only with the consumer market products.

## 1 INTRODUCTION

Firewire, known as IEEE 1394 standard high-performance serial bus or iLink, has been designed as a packet based computer bus in the new age[1]. The market of the Firewire system is just breaking especially in audio visual area. To use a new kind of media not only Firewire, we have to design and implement the device driver system and its associated environment including API. The Firewire specification considers that it is used for too many purposes, e.g., interface to storage devices, to audio visual equipments, to peripheral devices and between other computers. So it is difficult to categorize Firewire into legacy UNIX device types as either a network or a peripheral. In this paper, we present a Firewire driver development effort and discuss its way.<sup>1</sup>

<sup>1</sup>The device driver we mentioned here can be obtained from following URL:  
<ftp://ftp.uec.ac.jp/pub/firewire>

## 2 Overview of Firewire System

### 2.1 Firewire System

IEEE 1394 standard high-performance serial bus, often called "Firewire" or "iLink", is a standard interface designed to meet a lot of requirements of the new generation. The standard covers the whole system of Firewire from its physical layer to the layer of network management function. Firewire is capable of high network bandwidth; 100, 200, 400, 800, 1600 and 3200Mbps, permits connecting their heterogeneous bandwidth equipment on the same system, supports hot plug-in and -out on its working environment, and also provides both best effort and bandwidth guaranteed communication within one network media. From these advanced features, Firewire has the potential to integrate into only one bus system every peripheral interface of computers as SCSI, every network interface as Ethernet, every processor bus system as VME, and processor interconnect on multi processor systems.

In the IEEE 1394 standard, two types of media are defined as physical layer devices, i.e., backplane environment and cable environment. Today's market supplies only cable environment, while the products of backplane environment have not appeared yet. So, hereafter we only mention cable environment IEEE 1394 system.

Typical Firewire system consists of device nodes and cables, and its network topology is a tree structure. 2<sup>48</sup> memory space, including the Firewire specific control registers, is assigned to each device and

all the devices connected are mapped into unified memory space. One Firewire network can be connected up to  $(2^6 - 1 = 63)$  device nodes. When using a Firewire network bridge device, up to  $(2^{10} - 1 = 1023)$  network can be interconnected. Therefore, in the specification, almost  $2^{16}$  device nodes are allowed on a single system and every device is mapped into  $2^{64}$  memory space. Every communication action is brought with  $125 \mu$  second (8kHz) time slice whose value corresponds to the fairness unit in the Firewire system. The time slice unit is also divided into 6144 time slot. By allocating the number of the slot to each application, the network resource on the bandwidth guaranteed communication is managed. Three data transfer modes are defined i.e., Asynchronous request, Isochronous stream, and Asynchronous stream. Asynchronous request provides the function to communicate between the devices specifying the address in the packet with the memory accessing action, e.g., read, write, and transaction update same as memory devices. Isochronous stream mode provides the bandwidth guaranteed communication way using broadcast-like transmission not sent to specific devices. The stream has a feature of 6 bits channel identifier in the packet header. Asynchronous stream mode is defined in IEEE 1394.a standard, a supplemental specification of the original. This mode provides the best effort basis communication in a broadcast style. Due to so many requirements from today's media as bandwidth resource control, hot plug in/out, lower jitter transmission, and a large number of devices on one system, the whole specification of the Firewire became a large and complicated one. To support every function defined, a lot of effort is required compared with other network media.

IEEE 1394 standard series just defined a raw packet level communication protocol. When application uses Firewire, higher level protocols are also required such as IP over 1394, IEC61883 for real-time communication protocol, or SBP-2 for applying SCSI function[2, 3]. The standardization processes for their protocols have been accelerated in recent years.

The Firewire device driver has been developed for the major operating systems, such as Window 98, Windows NT and Linux, by distributing the Firewire

equipments.

## 2.2 IP networking

IP over Firewire standardization effort has been proceeded in IP1394 working group in IETF[4]. Although the fundamental architecture such as the selection of the transfer mode in each occasion has been reached consensus, the status of its standard is still on a draft stage and details of architecture will possibly be changed.

Two transmission modes, asynchronous request and asynchronous stream, are adopted by IP1394. In the case of IP unicist and ARP response, asynchronous request packet will be sent to the specific node. In other communication modes such as IP multicast, broadcast, and ARP request, asynchronous stream mode will be used. The stream channel assignment protocol are also defined.

1500 bytes value is adopted for the MTU size for IP packet on the IP over Firewire standard. On the lower speed communication modes such as 100 and 200Mbps, this 1500 bytes MTU is large compared with the link level MTU size in the Firewire network. Moreover, a Firewire device must ensure communication, even when the device only has poor input buffer compared with the link level MTU. The size of link level MTU on the Firewire may be more restricted and changed in some condition. If the smaller MTU issue in the intermediate link prove with the IP fragmentation, it leads the end-to-end network throughput to a poor one. Because a loss of fragmented packet causes unused packet sent. IP1394 group solved this issue by the link fragment mechanism; i.e., the fragmentation and assembly of the packet larger than MTU is done within the local media.

## 2.3 A/V device

Firewire is adopted as the standard of digital interface for audio visual device, since it provides lower jitter, bandwidth guaranteed transmission within the network level, and also provides lower equipping cost. Today, Firewire applications are one of the most popular concerns for the audio visual products such as various types of video camcorders and VCRs. The

protocol structure of the audio visual interface consists of 2 layers. The lower layer corresponds to a generic communication protocol for real-time stream media using isochronous stream. The higher one is designed as an adaptation protocol to each media type as NTSC, PAL, HDTV, MPEG and MIDI[5]. Some of these protocols are well known as DV video and the protocol set is approved as an IEC standard, IEC61883. In the real-time media system connected with the packet based network, the jitter of packets is the most important factor. If a larger jitter of packet is accepted, the system becomes more expensive because of the necessity of larger input buffer for absorbing the fluctuation. To avoid the problem, the specification requires strict timing in the order of  $\mu$  second. To support such strict real-time media packet timing, the computer system is recommended to take account of the hardware level not only the software.

## 2.4 Peripherals

SBP (Serial Bus Protocol) is designed as a SCSI adaptation protocol to Firewire and its standardization process is ongoing in ANSI. SBP is expected as the most major protocol for a storage device interface. However, the standards for computer peripheral are already established, e.g., SCSI and IDE apart from audio visual world. The Firewire application in computer peripherals has not been popular yet, even if Firewire has some advantages.

AV/C protocol is an audio visual device control protocol supposing the cooperation with such as DV video mentioned above, and provides audio visual equipment specific protocol set as "Play", "Rewind", and "Record". AV/C protocol only uses the raw Firewire functions in a simple way without any legacy protocol adaptation such as SCSI.

## 3 Integrating BSD system

The goal of our effort is to provide the Firewire environment on BSD system independent of the Firewire hardware. The device driver we developed is divided into two parts, i.e., the common part of the Firewire

system and the device dependent part. We describe both of them in this section. We implement some feature of the Firewire into the kernel level due to the indispensable to use it. The new kernel implemented feature concerning native Firewire are Bus Manager, Isochronous Resource Manager, and CSR register includes related functions,

### 3.1 Native Firewire socket

The driver supports native mode Firewire socket, or N1394. The most part of N1394 is written referring to NATM code that is ATM native mode socket system included from FreeBSD 3.0 release[6]. Both Asynchronous request and Isochronous communication modes can be used. The N1394 socket structure is shown in Fig. 1.

Unused fields in Fig 1, `sn1394_lch`, `sn1394_ltag`, and `sn1394_mode` remain due to backward compatibility previously released.

#### Isochronous stream

This mode provides the function for sending and receiving isochronous packet at a little programming cost. In this mode, user creates communication end point of socket, connects it to the isochronous channel specified by the N1394 socket structure, and communicates in the conventional datagram manner as UDP(Fig. 2).

Our Pentium MMX 233Mhz computer can receive the stream of consumer DV video, whose bandwidth is about 28Mbps using the isochronous interface. Although it also has a enough performance to send the video stream, the video device connected with Firewire cannot display due to the the reason of described later.

#### Asynchronous request

This mode provides asynchronous request function. In this mode, user creates communication end point of socket, connects it, sends the asynch packet user prepared, and waits for the response from the target device(Fig. 3). When using this mode, user must ensure the data structure of the asynchronous request

```

struct sockaddr_n1394 {
    u_int8_t    sn1394_len;           /* length           */
    u_int8_t    sn1394_family;        /* AF_N1394         */
    char        sn1394_if[IFNAMSIZ]; /* interface name    */
    u_int8_t    sn1394_fch;           /* isochronous channel */
    u_int8_t    sn1394_lch;           /* parameter unused  */
    u_int8_t    sn1394_ftag;          /* tag of isochronous */
    u_int8_t    sn1394_ltag;          /* parameter unused  */
    u_int8_t    sn1394_mode;          /* parameter unused  */
    u_int8_t    sn1394_flags;         /* specify socket status */
    u_int8_t    sn1394_spd;           /* transmission speed. */
};

```

Figure 1: N1394 socket structure

```

....
#include    <net1394/netfw.h>
...
main(){
    int s;
    char ifname[] = "lynx0";
    struct sockaddr_n1394 sfw;
    u_long recvbuf[512/4];
    int i, len;

    if((s = socket(AF_N1394, SOCK_DGRAM, PROTO_N1394ISO)) < 0 ){
        perror("socket");
        exit(1);
    }

    sfw.sn1394_family = AF_N1394;
    sfw.sn1394_lch = 63;
    sfw.sn1394_ltag = 0x1;
    bcopy(ifname, sfw.sn1394_if, IFNAMSIZ);

    if(connect(s, (struct sockaddr *)&sfw, sizeof(sfw)) < 0){
        perror("connect");
        exit(1);
    }
    while(1){
        len = recv(s, recvbuf, 512, 0);
        .....
    }
}

```

Figure 2: Code example to use Isochronous stream with socket

packet oneself in a different way from the isochronous case.

### 3.2 Firewire BSD and IP

Our implementation of IP over Firewire is partially compliance to the specification described in `draft-ietf-ip1394-08.txt`. The part of difference from the original are:

- Isochronous stream mode is used instead of Asynchronous stream in ARP and IP broadcast
- MCAP function is not implemented.

Firewire is a broadcast capable media and IP over Firewire adopts broadcast based ARP function same as Ethernet. So, the usage of the IP adaptation of Firewire does not require special mechanism. User can use IP network with following familiar operation:

```
# ifconfig lynx0 10.0.0.1 netmask 255.255.255.0
```

The improvement of the IP implementation is now stopping. So, our implementation does not accord with the latest IP1394 draft. However, since the basis of the specification has not been changed, the modification to accord with the latest one may not be a difficult.

### 3.3 Audio Visual device support

IEC61883 real-time media protocol requires strict packet timing in the order of  $\mu$  second. This timing condition can be satisfied with transmitting the isochronous packets at every 125  $\mu$  second time slice. Almost all Firewire devices support such a transmission with its programmable DMA function. Moreover, the protocol requires to write 24.576 MHz clock based time stamp into the specific packet header for the clock synchronization between the sender and the receivers. The generic BSD socket system cannot send packets as to satisfy these strict timing and cannot support the special treatment of the time stamp.

We prepare another device system for play out real-time stream media in addition to a network socket

interface, which is named "dv". The purpose of "dv" device is mainly for play out use, since the packet timing issue is not serious in the receiving. The "dv" device system treats a bunch of isochronous packet sequence as a minimum transmission unit. Every transmission operation on the device must be done with the bunch of packets. This treatment reduces the interrupt occurring at the end of DMA operation compared with the socket system if the number of packets in a bunch is large. To avoid the fluctuation in the system, the output queue is implemented and the length of the queue can be changed up to 12. The application and the kernel only copy the bunch to the DMA data entry and kicks a DMA action after the previous queued data is sent out. When the driver kicks a DMA action, only the first packet's time stamp field in the bunch is overwritten by a certain value. Then, the application must arrange the packet requiring the time stamp to become the first in the packet group. On the DV video format, the unit of bunch corresponds to the one video picture frame.

The "dv" device system provides two methods for stream packet play out. One is to use `write()` system call and the other is `mmap()` system call. Since both system calls do not kick the DMA operation itself, the application must tell the start of the operation to the kernel using `LYNX_DV_TXSTART` ioctl.

#### Use of `write()` system call

This mode is conventional use of "dv" device. This mode provides isochronous stream transmission with low program cost. In this mode, user open the device, load the data into a buffer and kick the DMA action (See `LYNX_DV_TXSTART`). This system call does not return the success code until the system comes by an empty buffer after sending out the previous data. This function satisfies the strict packet transmitting timing only at the kernel level. So, user will not care about the strict timing issue and it provides easiest way to make isochronous stream. When using this mode, user must be ensured that each packet is stored into 512 bytes boundaries in the output buffer and the first quadrat data of every packet corresponds to the header of the isochronous packet including the packet

```

....
#include      <net1394/netfw.h>
...
main(){
    int s;
    char ifname[] = "lynx0";
    struct sockaddr_n1394 sfw;
    u_long sendbuf[512/4], recvbuf[512/4];
    unsigned long long addr;
    int i, len;

    sfw.sn1394_family = AF_N1394;

    bcopy(ifname, sfw.sn1394_if, IFNAMSIZ);

    if((s = socket(AF_N1394, SOCK_DGRAM, PROTO_N1394ASY)) < 0 ){
        perror("socket");
        exit(1);
    }

    if(connect(s, (struct sockaddr *)&sfw, sizeof(sfw)) < 0){
        perror("connect");
        exit(1);
    }

    /*
    * Offset 0x0001ffff00000018 represents destination network/node/address
    * network = 0x0,
    * node    = 0x1,
    * address = 0xffff00000018(SPLIT_TIME_OUT_HI register)
    */
    addr = 0x0001ffff00000018ull;

    /* To make a packet for Read request for data quadret */
    sendbuf[0] = htonl(0x00000040 | ((addr >> 32) & 0xffff0000 ));
    sendbuf[1] = htonl(((addr >> 32) & 0x0000ffff ));
    sendbuf[2] = htonl(addr & 0xffffffff);

    send(s, sendbuf, 12, 0);

    /* Wait for a response from destination */
    len = recv(s, recvbuf, 512, 0);
    .....
}

```

Figure 3: Code example to use Asynchronous request with socket

size, tag, channel number, tcode and sync information(Fig. 4). The `nbytes` field of `write()` specifies the total size of the isochronous packets bunch.

### Use of `mmap()` system call

This mode provides the accessing way of the "dv" device buffer both for sending and for receiving allocated in the kernel space. In the mapped area of "dv" device, the control entry is placed before the memory space of output/input buffer. When using this mode, user manipulates the control entry in the "dv" specific manner additionally to the buffer management on the write system call. The data structure of the mapped "dv" device is shown in Fig. 5. In this mode, user open the device, map the "dv" device into the application space, access the space as conventional memory and kick the DMA actions(See `LYNX_DV_TXSTART/LYNX_DV_RXSTART`)(Fig. 6).

The size of both output/input buffers above is calculated as following:

$$size = sizeof(u\_int32\_t) \times N_q \times L_p \times N_p$$

where  $N_q$ ,  $L_p$  and  $N_p$  represent the number of attached queue, the size of the maximum packet, and the number of packets included in the buffer respectively. In our driver,  $N_q$ ,  $L_p$  and  $N_p$  are set to permanent value as 16, 512 and 300 respectively.

Also, `ioctl()` supports the following functions.

- `LYNX_DV_TXSTART`, `LYNX_DV_TXSTOP`  
Start and stop the DMA action for transmitting
- `LYNX_DVRX_START`, `LYNX_DVRX_STOP`  
Start and stop the DMA action for receiving
- `LYNX_SSignal_Wr`, `LYNX_GSignal_Wr`  
Set/get the process to raise signals, when transmitting DMA action is finished.
- `LYNX_GSignal_Rd`, `LYNX_GSignal_Rd`  
Set/get the process to raise signal, when receiving buffer is filled.
- `LYNX_GFrameSize`  
Get the amount of size of memory the device attached
- `LYNX_DV_SYNC`  
Set the queue length of transmission buffer. If the fluctuation factor is large in transmitting, a large

value should be settled.

We have released an DV application software that transmits DV video over IP using the function[8]. This application accomplishes DV quality communication only using the equipment on the consumer market. In SC98, we have presented the DV quality video communication between US and Japan with Transpac, Startap, and vBNS links. The application consumes over 30Mbps bandwidth including IP packet header in NTSC and it is not feasible to use such a bandwidth eater now. However, we believe that such volumes of bandwidth will be obtained easily in the near future, because the improvement of the high-speed link technology is too fast.

### 3.4 Supported Device

Our device driver supports two types of the Firewire chipset, i.e., Texas Instrument's PCILynx and Adaptec AIC5800[7, ?]. Since AIC5800 is derived from Apple Firefire chipset, the chipsets of the same series, e.g, Sony's chipset may work with a bit modification of the device driver code. Both chipsets only support the speed up to 200Mbps due to earlier product and does not support Asynchronous stream transmission defined in IEEE 1394.a itself. On PCILynx driver, our driver performs about 30Mbps TCP transmission performance with 100Mbps mode in netperf. This value shows that our driver has good performance compared with theoretical limit as 32Mbps. In some cheap PC configuration, the DMA operation stops in failure probably due to the poor PCI bus performance.

We are planning to develop the driver code for second generations's chipset as OHCI and PCILynx2 that supporting up to 400Mbps, asynchronous stream mode.

## 4 Conclusion

We developed a Firewire device driver on the FreeBSD system. Of course, the API specification and the driver we presented in this paper is not complete. It is still an open issue which type of UNIX

```

....
#include <machine/lynx.h>
...
main(){
    int d, qlen, dummy, frame;

#define MAXFRAME 12

    u_long sendbuf[MAXFRAME][512/4*300];
    u_long datalen[MAXFRAME];

    d = open("/dev/dv0", O_RDWR);

/* change queue size */
if( ioctl(d, LYNX_DV_SYNC, &qlen) < 0 ) {
    err(1, "LYNX_DV_SYNC");
}

/* kick playout */
if( ioctl(d, LYNX_DV_TXSTART, &dummy) < 0 ) {
    err(1, "LYNX_DV_TXSTART");
}

.....

frame = 0;
while(1){
frame++; if( frame == MAXFRAME ) frame = 0;

    ....

    if( write(d, sendbuf[frame], datalen[frame] * 512) < 0 ){
        perror("write");
    }
}

.....
}

```

Figure 4: Code example to use `write()` on "dv"

```

struct dv_data{
    /* Maximum size of output queue */
    u_int32_t n_write;
    /* Buffer now user locking, device does not send this data */
    u_int32_t a_write;
    /* Buffer now kernel locking, user must not access this data */
    u_int32_t k_write;
    u_int32_t write_done;
    /* Number of valid packet data in the buffer */
    u_int32_t write_len[16];
    /* Offset of buffer for writing */
    u_int32_t write_off[16];
    /* Maximum size of input queue */
    u_int32_t n_read;
    /* Buffer now user locking, device does not write this data */
    u_int32_t a_read;
    /* Buffer now kernel locking, this entry is not stable */
    u_int32_t k_read;
    u_int32_t read_done;
    /* Number of valid packet data in the buffer */
    u_int32_t read_len[16];
    /* Offset of buffer for reading */
    u_int32_t read_off[16];
};
...
isochronous data buffers for output
...
isochronous data buffers for input
...

```

Figure 5: Control entry on “dv”

```

....
#include <machine/lynx.h>
...
main(){
int d, size;
    u_int32_t *dvdata;
    d = open("/dev/dv0", O_RDWR);

/* obtain the amount of maximum buffer size incl. control, output and input */
    if(ioctl(d, LYNX_GFRAMESIZE, &size)){
        exit(1);
    }

    if((dvdata = (u_int32_t *)mmap((caddr_t) 0, size,
PROT_WRITE|PROT_READ, 0, d, (off_t)0)) < 0 ){
        err(1, "mmap");
    }
}
.....
/* Manipulate output/input buffer */
}

```

Figure 6: Code example to use `mmap()` on "dv"

system call accords with Firewire programming. Especially, the socket implementation of asynchronous request must be reconsidered. Because it is not a light programming effort to satisfy various mode of the asynchronous requests function. Many DV video equipment cannot display complete picture probably due to packet timing, even if using our "dv" implementation. A complete video picture appears in the limited configuration only. We are trying to solve the problem investigating the packet behaviour. The device having a Firewire interface will be more distributed and a variety of devices will increase. However, our developing capacity is limited. So, we want to cooperate with other UNIX development effort, e.g, implementing SBP with SCSI experts.

There is also other device driver development efforts on LINUX system. And, some UNIX vender announces the Firewire support on its OS as SGI IRIX. We are planning to make compatibility with other implementations, to reduce porting effort between UNIX systems.

## References

- [1] IEEE Computer Society, "IEEE Standard for a High Performance Serial Bus", IEEE Std. 1394(1996)
- [2] International Electrotechnical Commission, "Consumer audio/video equipment Digital interface", IEC 61883(1998)
- [3] American National Standard for Information systems, "Serial Bus Protocol 2", ANSI NCITS 325(1998)
- [4] P. Johanson, "IPv4 over IEEE1394", Internet Draft draft-ietf-ip1394-ipv4-8.txt(1998)
- [5] HD Digital Video Conference, "Specifications of Consumer-Use Digital VCRs using 6.3mm magnetic tape"(1995)
- [6] C. D. Cranor, "Integrating ATM Networking into BSD", See <http://www.ccrc.wustl.edu/pub/chuck/>

- [7] Texas Instruments, "1394 to PCI Bus Interface/TSB12LV21APGF Functional Specification" (1998)
- [8] Akimichi Ogawa *et al.*, "*Design and implementation of DV Stream over Internet*", *Proc. of Internet Work Shop 99(IWS99)* (1999)



## Newconfig: a dynamic-configuration framework for FreeBSD

Atsushi Furuta  
Software Research Associates, Inc.  
<furuta@sra.co.jp>

Jun-ichiro Hagino  
Research Laboratory, Internet Initiative Japan  
Inc.  
<itojun@itojun.org>

June 9, 1999

## Overview

- What is newconfig?
- Motivations
- Design
- Implementation
- Future work
- New-bus vs. newconfig
- Conclusion

## What is newconfig?

- Originally developed by Chris Torek in 4.4BSD.
- NetBSD, OpenBSD, BSD/OS
- We port this framework to FreeBSD-current

## Motivations

- PAO development
- CardBus support (hybrid of PCMCIA and PCI)
- There is no IRQ abstraction.
- There is no way to give configuration hint to PCI devices. (such as PCIC on PCI bus)

## Design

- A target is to merge to FreeBSD-current
  - ⇒ develop based on FreeBSD-current
- To implement dynamic configuration.
  - ⇒ add feature to handle kernel device driver tree dynamically.
- Support any drivers and any buses.

## Design (continued)

To remove old config ...

- Old auto-configuration mechanism appeared in 4.1BSD.
- config(8) is bus/machine dependent
- config(8) knows bus structure

## Design (continued)

Configuration hint and Plug-and-play bus

- No need configuration hint on PnP bus.
  - ... if every devices keep the PnP spec.
- Many rotten devices (or BIOSes).
- Device framework should provide "overriding" configuration hint.
  - ... or driver writers tend to "hard coding"

## Design (continued)

To support separation bus-dependent part

```
foo.c      foo driver core
foo_isa.c  foo driver ISA dependent part
foo_pci.c  foo driver PCI dependent part
```

**Experience** AMD 53C974 PCscsi controller.

⇒ 3-days by a beginner of driver programming.

## Design (continued)

Our idea of auto-configuration requirements:

1. Configuration hint information to device drivers
2. Bus/device hierarchy information
3. Inter-module dependency information
4. Device name → object file name mapping information

Newconfig handles all them, but static way.

## Implementation

config.new(8) reads 2 group of files

→ generate configuration data

→ statically linked to kernel

## Implementation (continued)

### 1. "files" file

- /usr/src/sys/conf/files.newconf
- /usr/src/sys/i386/conf/files.i386.newconf
- provided by programmer

### 2. CONFIG file

- /usr/src/sys/i386/conf/NEWCONF
- provided by user
- like "GENERIC", or "LINT"

## Future work

- Dynamic configuration for newconfig
- When are device configuration information given?
  - Compile time (static)
  - Boot time (a.k.a. UserConfig)
  - Run time (dynamic)
- The weakest point of the current implementation of newconfig.

## Future work (continued)

### Implementation of dynamic config

- A utility that parse "files" file for dynamic module.
- A module loader with parsed "files" file and config hint.
- Kernel codes that handles loaded config information and registers new bus/device hierarchy.

## New-bus vs. newconfig

### New-bus

<http://www.freebsd.org/~dfr/devices.html>

1. Configuration hint information to device drivers
2. Bus/device hierarchy information
3. Inter-module dependency information
4. Device name → object file name mapping information

## Conclusion

- Static vs. dynamic: both config required
- To provide the way of overriding PnP information
- Explicit syntax of configuration data are useful

## References

### newconfig information

<http://www.jp.freebsd.org/newconfig/>

### presentation draft

<http://home.jp.freebsd.org/~furuta/freenix/>

# The Vinum Volume Manager

Greg Lehey

*Nan Yang Computer Services Ltd.*

`grog@lemis.com`

## ABSTRACT

The *Vinum Volume Manager* is a block device driver which implements virtual disk drives. It isolates disk hardware from the block device interface and maps data in ways which result in an increase in flexibility, performance and reliability compared to the traditional slice view of disk storage. Vinum implements the RAID-0, RAID-1 and RAID-5 models, both individually and in combination.

## Introduction

Disk hardware is evolving rapidly, and the current UNIX disk abstraction is inadequate for a number of modern applications. In particular, file systems must be stored on a single disk partition, and there is no kernel support for redundant data storage. In addition, the direct relationship between disk volumes and their location on disk make it generally impossible to enlarge a disk volume once it has been created. Performance can often be limited by the maximum data rate which can be achieved with the disk hardware.

The largest modern disks store only about 50 GB, but large installations, in particular web sites, routinely have more than a terabyte of disk storage, and it is not uncommon to see disk storage of several hundred gigabytes even on PCs. Storage-intensive applications such as Internet World-Wide Web and FTP servers have accelerated the demand for high-volume, reliable storage systems which deliver high performance in a heavily concurrent environment.

## The problems

Various solutions to these problems have been proposed and implemented:

### Disks are too small

The *ufs* file system can theoretically span more than a petabyte ( $2^{50}$  or  $10^{15}$  bytes) of storage, but no current disk drive comes close to this size. Although the size problem is not as acute as it was ten years ago, there is a

simple solution: the disk driver can create an abstract device which stores its data on a number of disks. A number of such implementations exist, though none appear to have become mainstream.

### Access bottlenecks

Modern systems frequently need to access data in a highly concurrent manner. For example, the FTP server *wcarchive.cdrom.com* maintains up to 3,600 concurrent FTP sessions and has a 100 Mbit/s connection to the outside world, corresponding to about 12 MB/s.

Current disk drives can transfer data sequentially at up to 30 MB/s, but this value is of little importance in an environment where many independent processes access a drive, where they may achieve only a fraction of these values. In such cases it's more interesting to view the problem from the viewpoint of the disk subsystem: the important parameter is the load that a transfer places on the subsystem, in other words the time for which a transfer occupies the drives involved in the transfer.

In any disk transfer, the drive must first position the heads, wait for the first sector to pass under the read head, and then perform the transfer. These actions can be considered to be atomic: it doesn't make any sense to interrupt them.

Consider a typical transfer of about 10 kB: the current generation of high-performance disks can position the heads in an average of 6 ms. The fastest drives spin at 10,000 rpm, so the average rotational latency (half a revolution) is 3 ms. At 30 MB/s, the transfer itself takes about 350  $\mu$ s, almost nothing compared to the positioning time. In such a case, the effective transfer rate drops to a little over 1 MB/s and is clearly highly dependent on the transfer size.

The traditional and obvious solution to this bottleneck is “more spindles”: rather than using one large disk, it uses several smaller disks with the same aggregate storage space. Each disk is capable of positioning and transferring independently, so the effective throughput increases by a factor close to the number of disks used.

The exact throughput improvement is, of course, smaller than the number of disks involved: although each drive is capable of transferring in parallel, there is no way to ensure that the requests are evenly distributed across the drives. Inevitably the load on one drive will be higher than on another.

The evenness of the load on the disks is strongly dependent on the way the data is shared across the drives. In the following discussion, it's convenient to think of the disk storage as a large number of data sectors which are addressable by number, rather like the pages in a book. The most obvious method is to divide the virtual disk into groups of consecutive sectors the size of the individual physical disks and store them in this manner, rather like taking a large book and tearing it into smaller sections. This method is called *concatenation* and has the advantage that the disks do not need to have any specific size relationships. It works well when the access to the virtual disk is spread evenly about its address space. When access is concentrated on a smaller area, the improvement is less marked. Figure 1 illustrates the sequence in which storage units are allocated in a concatenated organization.

Disk 1	Disk 2	Disk 3	Disk 4
0	6	10	12
1	7	11	13
2	8		14
3	9		15
4			16
5			17

Figure 1: Concatenated organization

An alternative mapping is to divide the address space into smaller, even-sized components and store them sequentially on different devices. For example, the first 256 sectors may be stored on the first disk, the next 256 sectors on the next disk and so on. After filling the last disk, the process repeats until the disks are full. This mapping is called *striping* or RAID-0, though the latter term is somewhat misleading: it provides no redundancy. Striping requires somewhat more effort to locate the data, and it can cause additional I/O load where a transfer is spread over multiple disks, but it can

also provide a more constant load across the disks. Figure 2 illustrates the sequence in which storage units are allocated in a striped organization.

Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
20	21	22	23

Figure 2: Striped organization

## Data integrity

The final problem with current disks is that they are unreliable. Although disk drive reliability has increased tremendously over the last few years, of all the core components of a server they are still the most likely to fail. When they do, the results can be catastrophic: replacing a failed disk drive and restoring data to it can take days.

The traditional way to approach this problem has been *mirroring*, keeping two copies of the data on different physical hardware. Since the advent of the RAID levels, this technique has also been called *RAID level 1* or *RAID-1*. Any write to the volume writes to both locations; a read can be satisfied from either, so if one drive fails, the data is still available on the other drive.

Mirroring has two problems:

- The price. It requires twice as much disk storage as a non-redundant solution.
- The performance impact. Writes must be performed to both drives, so they take up twice the bandwidth of a non-mirrored volume. Reads do not suffer from a performance penalty: it even looks as if they are faster. This issue will be discussed in the “Performance issues” section.

An alternative solution is *parity*, implemented in the RAID levels 2, 3, 4 and 5. Of these, RAID-5 is the most interesting. As implemented in Vinum, it is a variant on a striped organization which dedicates one block of each stripe to parity of the other blocks. In RAID-5, the location of this parity block changes from one stripe to the next. Figure 3 shows the RAID-5 organization. The numbers in the data blocks indicate the relative block numbers.

Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	Parity
3	4	Parity	5
6	Parity	7	8
Parity	9	10	11
12	13	14	Parity
15	16	Parity	17

**Figure 3: RAID-5 organization**

Compared to mirroring, RAID-5 has the advantage of requiring significantly less storage space. Read access is similar to that of striped organizations, but write access is significantly slower, approximately 25% of the read performance. If one drive fails, the array can continue to operate in degraded mode: a read from one of the remaining accessible drives continues normally, but a read from the failed drive is recalculated from the corresponding block from all the remaining drives.

## Current implementations

The problems of size, performance and reliability have solutions that are only partially compatible. In particular, redundant data storage and performance improvements require different solutions and affect each other negatively.

The current trend is to realize such systems in *disk array* hardware, which looks to the host system like a very large disk. Disk arrays have a number of advantages:

- They are portable. Since they have a standard interface, usually SCSI, but increasingly also IDE, they can be installed on almost any system without kernel modifications.
- They have the potential to offer impressive performance: they offload the calculations (in particular, the parity calculations for RAID-5) to the array, and in the case of replicated data, the aggregate transfer rate to the array is less than it would be to local disks. Striping ("RAID-0") and RAID-5 organizations also spread the load more evenly over the physical disks, thus improving performance. Nevertheless, an array is typically connected via a single SCSI connection, which can be a bottleneck, and some implementations show surprisingly poor performance which cannot be explained by the hardware configuration. Installing a disk array does not guarantee better performance.

- They are reliable. A good disk array offers a large number of features designed to enhance reliability, including enhanced cooling, hot-plugging (the ability to replace a drive while the array is running) and automatic failure recovery.

On the other hand, disk arrays are relatively expensive and not particularly flexible. An alternative is a software-based *volume manager* which performs similar functions in software. A number of these systems exist, notably the VERITAS® volume manager [Veritas], Solaris *DiskSuite* [Solstice], IBM's *Logical Volume Facility* [IBM] and SCO's *Virtual Disk Manager* [SCO]. An implementation of RAID software is also available for Linux [Linux].

## Vinum

*Vinum* is an open source [OpenSource] volume manager implemented under FreeBSD [FreeBSD]. It was inspired by the VERITAS® volume manager and implements many of the concepts of VERITAS®. Its key features are:

- Vinum implements RAID-0 (striping), RAID-1 (mirroring) and RAID-5 (rotated block-interleaved parity). In RAID-5, a group of disks are protected against the failure of any one disk by an additional disk with block checksums of the other disks.<sup>1</sup>
- Drive layouts can be combined to increase robustness, including striped mirrors (so-called "RAID-10").
- Vinum implements only those features which appear useful. Some commercial volume managers appear to have been implemented with the goal of maximizing the size of the spec sheet. Vinum does not implement "ballast" features such as RAID-4. It would have been trivial to do so, but the only effect would have been to further confuse an already confusing topic.
- Volume managers initially emphasized reliability and performance rather than ease of use. The results are frequently down time due to misconfiguration, with consequent reluctance on the part of operational personnel to attempt to use the more unusual features of the product. Vinum attempts to provide an easier-to-use non-GUI interface.

1. The RAID-5 functionality is currently available under license from Cybernet, Inc. [Cybernet]. It will be released as open source at a later date.

## How Vinum addresses the Three Problems

As mentioned above, Vinum addresses three main deficiencies of traditional disk hardware. This section examines them in more detail.

### Vinum objects

In order to address these problems, vinum implements a four-level hierarchy of objects:

- The most visible object is the virtual disk, called a *volume*. Volumes have essentially the same properties as a UNIX disk drive, though there are some minor differences. They have no size limitations.
- Volumes are composed of *plexes*, each of which represent the total address space of a volume. This level in the hierarchy thus provides redundancy.
- Since Vinum exists within the UNIX disk storage framework, it would be possible to use UNIX partitions as the building block for multi-disk plexes, but in fact this turns out to be too inflexible: UNIX disks can have only a limited number of partitions. Instead, Vinum subdivides a single UNIX partition (the *drive*) into contiguous areas called *subdisks*, which it uses as building blocks for plexes.
- Subdisks reside on Vinum *drives*, currently UNIX partitions. Vinum drives can contain any number of subdisks. With the exception of a small area at the beginning of the drive, which is used for storing configuration and state information, the entire drive is available for data storage.

The following sections describe the way these objects provide the functionality required of Vinum.

### Volume size considerations

Plexes can include multiple subdisks spread over all drives in the Vinum configuration. As a result, the size of an individual drive does not limit the size of a plex, and thus of a volume.

### Redundant data storage

Vinum provides both mirroring and RAID-5. It implements mirroring by attaching multiple plexes to a volume. Each plex is a representation of the data in a volume. A volume may contain between one and eight plexes.

Although a plex represents the complete data of a volume, it is possible for parts of the representation to be physically missing, either by design (by not defining a subdisk for parts of the plex) or by accident (as a result of the failure of a drive). As long as at least one plex can provide the data for the complete address range of the volume, the volume is fully functional.

From an implementation standpoint, it is not practical to represent a RAID-5 organization as a collection of plexes. This issue is discussed below.

### Performance issues

By spreading data across multiple disks, Vinum can deliver much higher performance than a single disk. This issue will be discussed in more detail below.

### RAID-5

Conceptually, RAID-5 is used for redundancy, but in fact the implementation is a kind of striping. This poses problems for the implementation of Vinum: should it be a kind of plex or a kind of volume? It would have been possible to implement it either way, but it proved to be simpler to implement RAID-5 as a plex type. This means that there are two different ways of ensuring data redundancy: either have more than one plex in a volume, or have a single RAID-5 plex. These methods can be combined.

### Which plex organization?

Vinum implements only that subset of RAID organizations which make sense in the framework of the implementation. It would have been possible to implement all RAID levels, but there was no reason to do so. Each of the chosen organizations has unique advantages:

- Concatenated plexes are the most flexible: they can contain any number of subdisks, and the subdisks may be of different length. The plex may be extended by adding additional subdisks. They require less CPU time than striped or RAID-5 plexes, though the difference in CPU overhead from striped plexes is not measurable. On the other hand, they are most susceptible to "hot spots", where one disk is very active and others are idle.
- The greatest advantage of striped (RAID-0) plexes is that they reduce hot spots: by choosing an optimum sized stripe (empirically determined to be in the order of 256 kB), the load on the component

drives can be made more even. The disadvantages of this approach are (fractionally) more complex code and restrictions on subdisks: they must be all the same size, and extending a plex by adding new subdisks is so complicated that Vinum currently does not implement it. Vinum imposes an additional, trivial restriction: a striped plex must have at least two subdisks, since otherwise it is indistinguishable from a concatenated plex.

- RAID-5 plexes are effectively an extension of striped plexes. Compared to striped plexes, they offer the advantage of fault tolerance, but the disadvantages of higher storage cost and significantly higher overhead, particularly for writes. The code is an order of magnitude more complex than for concatenated and striped plexes. Like striped plexes, RAID-5 plexes must have equal-sized subdisks and cannot be extended. Vinum enforces a minimum of three subdisks for a RAID-5 plex, since any smaller number would not make any sense.

These are not the only possible organizations. In addition, the following could have been implemented:

- RAID-4, which differs from RAID-5 only by the fact that all parity data is stored on a specific disk. This simplifies the algorithms somewhat at the expense of drive utilization: the activity on the parity disk is a direct function of the read to write ratio. Since Vinum implements RAID-5, RAID-4's only advantage is nullified.
- RAID-3, effectively an implementation of RAID-4 with a stripe size of one byte. Each transfer requires reading each disk (with the exception of the parity disk for reads). Without spindle synchronization (where the corresponding sectors pass the heads of each drive at the same time), RAID-3 would be very inefficient. In a multiple-access system, it also causes high latency.

An argument for RAID-3 does exist where a single process requires very high data rates. With spindle synchronization, this would be a potentially useful addition to Vinum.

- RAID-2, which uses two subdisks to store a Hamming code, and which otherwise resembles RAID-3. Compared to RAID-3, it offers a lower data density, higher CPU usage and no compensating advantages.

In addition, RAID-5 can be interpreted in two different

ways: the data can be striped, as in the Vinum implementation, or it can be written serially, exhausting the address space of one subdisk before starting on the other, effectively a modified concatenated organization. There is no recognizable advantage to this approach, since it does not provide any of the other advantages of concatenation.

## Some examples

Vinum maintains a *configuration database* which describes the objects known to an individual system. Initially, the user creates the configuration database from one or more configuration files with the aid of the *vinum(8)* utility program. Vinum stores a copy of its configuration database on each drive under its control. This database is updated on each state change, so that a restart accurately restores the state of each Vinum object.

## The configuration file

The configuration file describes individual Vinum objects. The definition of a simple volume might be:

```
drive a device /dev/da3h
volume myvol
    plex org concat
        sd length 512m drive a
```

This file describes a four Vinum objects:

- The *drive* line describes a disk partition (*drive*) and its location relative to the underlying hardware. It is given the symbolic name *a*. This separation of the symbolic names from the device names allows disks to be moved from one location to another without confusion.
- The *volume* line describes a volume. The only required attribute is the name, in this case *myvol*.
- The *plex* line defines a plex. The only required parameter is the organization, in this case *concat*. No name is necessary: the system automatically generates a name from the volume name by adding the suffix *.px*, where *x* is the number of the plex in the volume. Thus this plex will be called *myvol.p0*.
- The *sd* line describes a subdisk. The minimum specifications are the name of a drive on which to store it, and the length of the subdisk. As with plexes, no name is necessary: the system automatically assigns names derived from the plex name by adding the suffix *.sx*, where *x* is the

number of the subdisk in the plex. Thus Vinum gives this subdisk the name *myvol.p0.s0*

This particular volume has no specific advantage over a conventional disk partition. It contains a single plex, so it is not redundant. The plex contains a single subdisk, so there is no difference in storage allocation from a conventional disk partition. The following sections illustrate various more interesting configurations.

### Increased resilience: mirroring

The resilience of a volume can be increased either by mirroring or by using RAID-5 plexes. When laying out a mirrored volume, it is important to ensure that the subdisks of each plex are on different drives, so that a drive failure will not take down both plexes. The following configuration mirrors a volume:

```
drive b device /dev/da4h
volume mirror
  plex org concat
    sd length 512m drive a
  plex org concat
    sd length 512m drive b
```

In this example, it was not necessary to specify a definition of drive *a* again, since Vinum keeps track of all objects in its configuration database.

In this example, each plex contains the full 512 MB of address space. As in the previous example, each plex contains only a single subdisk.

### Optimizing performance

The mirrored volume in the previous example is more resistant to failure than an unmirrored volume, but its performance is less: each write to the volume requires a write to both drives, using up a greater proportion of the total disk bandwidth. Performance considerations demand a different approach: instead of mirroring, the data is striped across as many disk drives as possible. The following configuration shows a volume with a plex striped across four disk drives:

```
drive c device /dev/da5h
drive d device /dev/da6h
volume stripe
  plex org striped 512k
    sd length 128m drive a
    sd length 128m drive b
    sd length 128m drive c
    sd length 128m drive d
```

As before, it is not necessary to define the drives which are already known to Vinum.

### Increased resilience: RAID-5

The alternative approach to resilience is RAID-5. A RAID-5 configuration might look like:

```
drive e device /dev/da6h
volume raid5
  plex org raid5 512k
    sd length 128m drive a
    sd length 128m drive b
    sd length 128m drive c
    sd length 128m drive d
    sd length 128m drive e
```

Although this plex has five subdisks, its size is the same as the plexes in the other examples, since the equivalent of one subdisk is used to store parity information.

On creation, RAID-5 plexes are in the *init* state: before they can be used, the parity data must be created. Vinum currently initializes RAID-5 plexes by writing binary zeros to all subdisks, though a probable future alternative is to rebuild the parity blocks, which allows better recovery of crashed plexes.

### Resilience and performance

With sufficient hardware, it is possible to build volumes which show both increased resilience and increased performance compared to standard UNIX partitions. Mirrored disks will always give better performance than RAID-5, so a typical configuration file might be:

```
volume raid10
  plex org striped 512k
    sd length 102480k drive a
    sd length 102480k drive b
    sd length 102480k drive c
    sd length 102480k drive d
    sd length 102480k drive e
  plex org striped 512k
    sd length 102480k drive c
    sd length 102480k drive d
    sd length 102480k drive e
    sd length 102480k drive a
    sd length 102480k drive b
```

The subdisks of the second plex are offset by two drives from those of the first plex: this helps ensure that writes do not go to the same subdisks even if a transfer goes over two drives.

### Creating file systems

Volumes appear to the system to be identical to disks, with one exception. Unlike UNIX drives, Vinum does not partition volumes, which thus do not contain a partition table. This has required modification to some

disk utilities, notably *newfs*, which previously tried to interpret the last letter of a Vinum volume name as a partition identifier. For example, a disk drive may have a name like */dev/wd0a* or */dev/da2h*. These names represent the first partition (a) on the first (0) IDE disk (wd) and the eighth partition (h) on the third (2) SCSI disk (da) respectively. By contrast, a Vinum volume might be called */dev/vinum/concat*, a name which has no relationship with a partition name.

Normally, *newfs(8)* interprets the name of the disk and complains if it cannot understand it. For example:

```
# newfs /dev/vinum/concat
newfs: /dev/vinum/concat: can't figure out
file system partition
```

In order to create a file system on this volume, use the *-v* option to *newfs(8)*:

```
# newfs -v /dev/vinum/concat
```

## Startup

Vinum stores configuration information on the disk slices in essentially the same form as in the configuration files. When reading from the configuration database, Vinum recognizes a number of keywords relating to object state which are not allowed in the configuration files. Vinum does not store information about drives in the configuration information: it finds the drives by scanning the configured disk drives for partitions with a Vinum label. This enables Vinum to identify drives correctly even if they have been assigned different UNIX drive IDs.

At system startup, Vinum reads the configuration database from one of the Vinum drives. Under normal circumstances, each drive contains an identical copy of the configuration database, so it does not matter which drive is read. After a crash, however, Vinum must determine which drive was updated most recently and read the configuration from this drive.

## Performance issues

At present only superficial performance measurements have been made. They show that the performance is very close to what could be expected from the underlying disk driver performing the same operations as Vinum performs: in other words, the overhead of Vinum itself is negligible. This does not mean that

Vinum has perfect performance: the choice of requests has a strong impact on the overall subsystem performance, and there are some known areas which could be improved upon. In addition, the user can influence performance by the design of the volumes.

The following sections examine some factors which influence performance.

**Note:** The performance measurements in this section were done on some very old pre-SCSI-1 disk drives. The absolute performance is correspondingly poor. The intention of the following graphs is to show relative performance, not absolute performance. Other tests indicate that the performance relationships also apply to modern high-end hardware.

## The influence of stripe size

In striped and RAID-5 plexes, the stripe size has a significant influence on performance. In all plex structures with more than one subdisk, the possibility exists that a single transfer to or from a volume will be remapped into more than one physical I/O request. This is never desirable, since the average latency for multiple transfers is always larger than the average latency for single transfers to the same kind of disk hardware. Spindle synchronization does not help here, since there is no deterministic relationship between the positions of the data blocks on the different disks. Within the bounds of the current BSD I/O architecture (maximum transfer size 128 kB) and current disk hardware, this increase in latency can easily offset any speed increase in the transfer.

In the case of a concatenated plex, this remapping occurs only when a request overlaps a subdisk boundary, which is seldom enough to be negligible. In a striped or RAID-5 plex, however, the probability is an inverse function of the stripe size. For this reason, a stripe size of 256 kB appears to be optimum: it is small enough to create a relatively random mapping of file system hot spots to individual disks, and large enough to ensure that 99% of all transfers involve only a single data subdisk. Figure 4 shows the effect of stripe size on read and write performance, obtained with *rawio* [rawio]. This measurement used eight concurrent processes to access volumes with striped plexes with different stripe sizes. The graph shows the disadvantage of small stripe sizes, which can cause a significant performance degradation even compared to a single disk.

## The influence of RAID-1 mirroring

Mirroring has different effects on read and write throughput. A write to a mirrored volume causes writes to each plex, so write performance is less than for a non-mirrored volume. A read from a mirrored volume, however, reads from only one plex, so read performance can improve.

There are two different scenarios for these performance changes, depending on the layout of the subdisks comprising the volume. Two basic possibilities exist for a mirrored, striped plex.

### One disk per subdisk

The optimum layout, both for reliability and for performance, is to have each subdisk on a separate disk. An example might be the following configuration, similar to the sample "RAID-10" configuration seen above.

```
volume raid10
  plex org striped 512k
    sd length 102480k drive a
    sd length 102480k drive b
    sd length 102480k drive c
    sd length 102480k drive d
  plex org striped 512k
    sd length 102480k drive e
    sd length 102480k drive f
    sd length 102480k drive g
    sd length 102480k drive h
```

In this case, the volume is spread over a total of eight disks. This has the following effects:

- Read access: by default, read accesses will alternate across the two plexes, giving a performance improvement close to 100%.
- Write access: writes must be performed to both disks, doubling the bandwidth requirement. Since the available bandwidth is also double, there should be little difference in throughput.

At present, due to lack of hardware, no tests have been made of this configuration.

### Both plexes on the same disks

An alternative layout is to spread the subdisks of each plex over the same disks:

```
volume raid10
  plex org striped 512k
    sd length 102480k drive a
    sd length 102480k drive b
    sd length 102480k drive c
    sd length 102480k drive d
  plex org striped 512k
```

```
sd length 102480k drive c
sd length 102480k drive d
sd length 102480k drive a
sd length 102480k drive b
```

In this configuration, the subdisks covering a specific plex address space have been placed on different drives, thus improving both performance and resilience. This configuration has the following properties:

- Read access: by default, read accesses will alternate across the two plexes. Since there is no increase in bandwidth, there will be little difference in performance through the second plex.
- Write access: writes must be performed to both disks, doubling the bandwidth requirement. In this case, the bandwidth has not increase, so write throughput will decrease by approximately 50%.

Figure 4 also shows the effect of mirroring in this manner. The results are very close to the theoretical predictions.

## The influence of request size

As seen above, the throughput of a disk subsystem is the sum of the latency (the time taken to position the disk hardware over the correct part of the disk) and the time to transfer the data to or from the disk. Since latency is independent of transfer size and much larger than the transfer time for typical transfers, overall throughput is strongly dependent on the size of the transfer, as Figure 5 shows. Unfortunately, there is little that can be done to influence the transfer size. In FreeBSD, it tends to be closer to 10 kB than to 30 kB.

## The influence of concurrency

Vinum aims to give best performance for a large number of concurrent processes performing random access on a volume. Figure 6 shows the relationship between number of processes and throughput for a raw disk volume and a Vinum volume striped over four such disks with between one and 128 concurrent processes with an average transfer size of 16 kB. The actual transfers varied between 512 bytes and 32 kB, which roughly corresponds to ufs usage.

This graph clearly shows the differing effects of multiple concurrent processes on the Vinum volume and the relative lack of effect on a single disk. The single disk is saturated even with one process, while Vinum shows a continual throughput improvement with up to 128 processes, by which time it has practically leveled off.

## The influence of request structure

For concatenated and striped plexes, Vinum creates request structures which map directly to the user-level request buffers. The only additional overhead is the allocation of the request structure, and the possibility of improvement is correspondingly small.

With RAID-5 plexes, the picture is very different. The strategic choices described above work well when the total request size is less than the stripe width. By contrast, it does not perform optimally when a request is larger than the size of all blocks in a stripe. The requests map to contiguous space on the disk but non-contiguous space in the user buffer. An optimal implementation would perform one I/O request per drive and map to the user buffer in software. By contrast, Vinum performs separate I/O requests for each stripe.

In practice, this inefficiency should not cause any problems: as discussed above, the optimum stripe size is larger than the maximum transfer size, so this situation arises only when an inappropriately small stripe size is chosen.

Figure 7 shows the RAID-5 tradeoffs:

- The RAID-5 write throughput is approximately half of the RAID-1 throughput in figure 4, and one-quarter of the write throughput of a striped plex.
- The read throughput is similar to that of striped volume of the same size.

Although the random access performance increases continually with increasing stripe size, the sequential access performance peaks at about 20 kB for writes and 35 kB for reads. This effect has not yet been adequately explained, but may be due to the nature of the test (8 concurrent processes writing the same data at the same time).

## Availability

Vinum is available without RAID-5 functionality under a Berkeley-style copyright as part of the FreeBSD 3.1 distribution. It is also available at [vinum]. The RAID-5 functionality is available under licence from Cybernet, Inc. [Cybernet], and is included in their *NetMAX* Internet connection package.

## Future directions

The current version of Vinum implements the core functionality. A number of additional features are under consideration:

- *Hot spare* capability: on the failure of a disk drive, the volume manager automatically recovers the data to another drive.
- *Logging* changes to a degraded volume. Rebuilding a plex usually requires copying the entire volume. In a volume with a high read to write ratio, if a disk goes down temporarily and then becomes accessible again (for example, as the result of controller failure), most of the data is already correct and does not need to be copied. Logging pinpoints which blocks require copying in order to bring the stale plex up to date.
- *Snapshots* of a volume. It is often useful to freeze the state of a volume, for example for backup purposes. A backup of a large volume can take several hours. It can be inconvenient or impossible to prohibit updates during this time. A snapshot solves this problem by maintaining *before images*, a copy of the old contents of the modified data blocks. Access to the plex reads the blocks from the snapshot plex if it contains the data, and from another plex if it does not.

Implementing snapshots in Vinum alone would solve only part of the problem: there must also be a way to ensure that the data on the file system is consistent from a user standpoint when the snapshot is taken. This task involves such components as file systems and databases and is thus outside the scope of Vinum.

- A *SNMP interface* for central management of Vinum systems.
- A *GUI* interface is currently *not* planned, though it is relatively simple to program, since no kernel code is needed. As the number of failures testify, a good GUI interface is apparently very difficult to write, and it tends to gloss over important administrative aspects, so it's not clear that the advantages justify the effort. On the other hand, a graphical output of the configuration could be of advantage.

- An *extensible ufs*. It is possible to extend the size of some modern file systems after they have been created. Although ufs (the *UNIX File System*, previously called the *Berkeley Fast File System*) was not designed for such extension, it is trivial to implement extensibility. This feature would allow a user to add space to a file system which is approaching capacity by first adding subdisks to the plexes and then extending the file system.
- *Remote data replication* is of interest either for backup purposes or for read-only access at a remote site. From a conceptual viewpoint, it could be achieved by interfacing to a network driver instead of a local disk driver.
- *Extending striped and RAID-5 plexes* is a slow complicated operation, but it is feasible.

[products/system/disksuite.html](http://www.sun.com/products/system/disksuite.html)

[veritas] The VERITAS® Volume manager,  
<http://www.veritas.com/product-info/vm/index.htm>.

[vinum] Greg Lehey, *The Vinum Volume Manager*,  
<http://www.lemis.com/vinum.html>. An extended version of this paper.

[Wong] Brian Wong, *RAID: What does it mean to me?*,  
SunWorld Online, September 1995.  
<http://www.sunworld.com/sunworldonline/swol-09-1995/swol-09-raid5.html>

## References

[CMD] CMD Technology, Inc June 1993, *The Need For RAID, An Introduction*.  
<http://www.fdma.com/info/raidinto.html>

[Cybernet] *The NetMAX Station*,  
<http://www.cybernet.com/netmax/index.html>. The first product using the Vinum Volume Manager.

[FreeBSD] FreeBSD home page,  
<http://www.FreeBSD.org/>

[IBM] *AIX Version 4.3 System Management Guide: Operating System and Devices, Logical Volume Storage Overview*  
[http://www.austin.ibm.com/doc\\_link/en\\_US/a\\_doc\\_lib/aixbman/baseadm/lvm\\_overview.htm](http://www.austin.ibm.com/doc_link/en_US/a_doc_lib/aixbman/baseadm/lvm_overview.htm)

[Linux] *Logical Volume Manager for Linux*,  
<http://linux.msede.com/lvm/>.

[McKusick] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*, Addison Wesley, 1996.

[OpenSource] *The Open Source Page*,  
<http://www.opensource.org/>

[rawio] A raw disk I/O benchmark.  
<ftp://ftp.lemis.com/pub/rawio.tar.gz>

[SCO] *SCO Virtual Disk Manager*,  
<http://www.sco.com/products/layered/ras/virtual.html>.

[Solstice] <http://www.sun.com/solstice/em->

## Illustrations

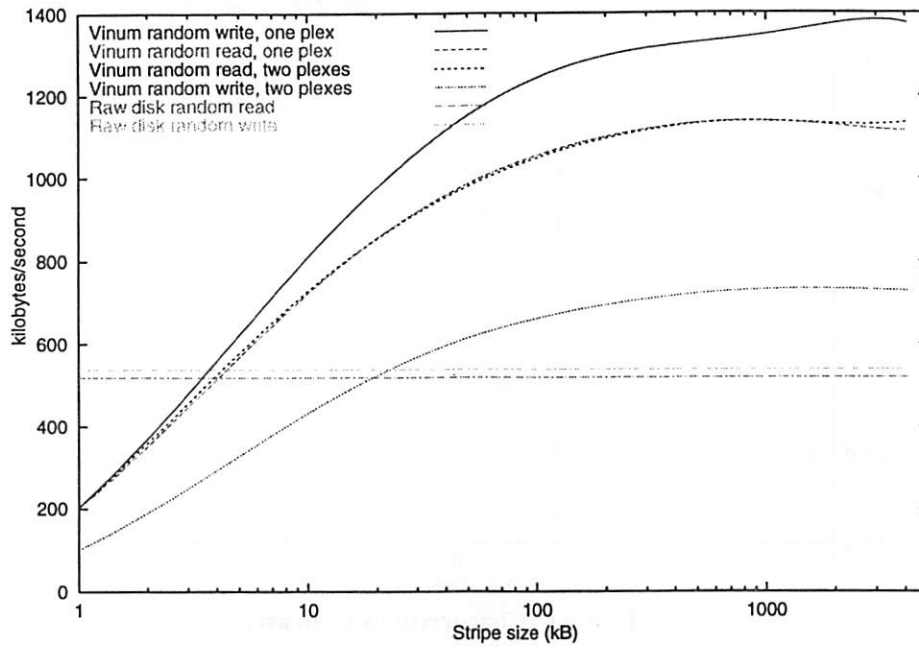


Figure 4: The influence of stripe size and mirroring

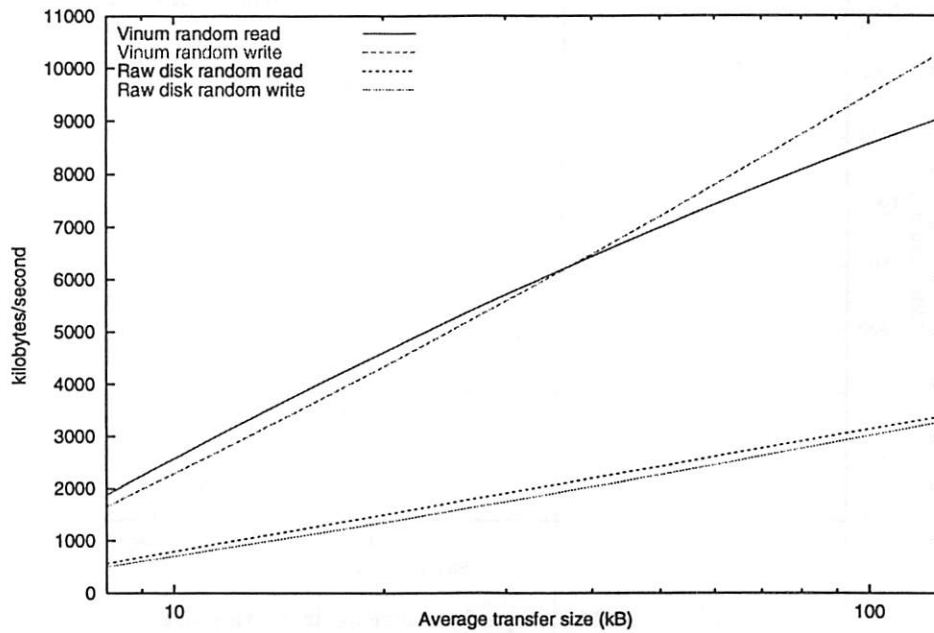


Figure 5: Throughput as function of transfer size

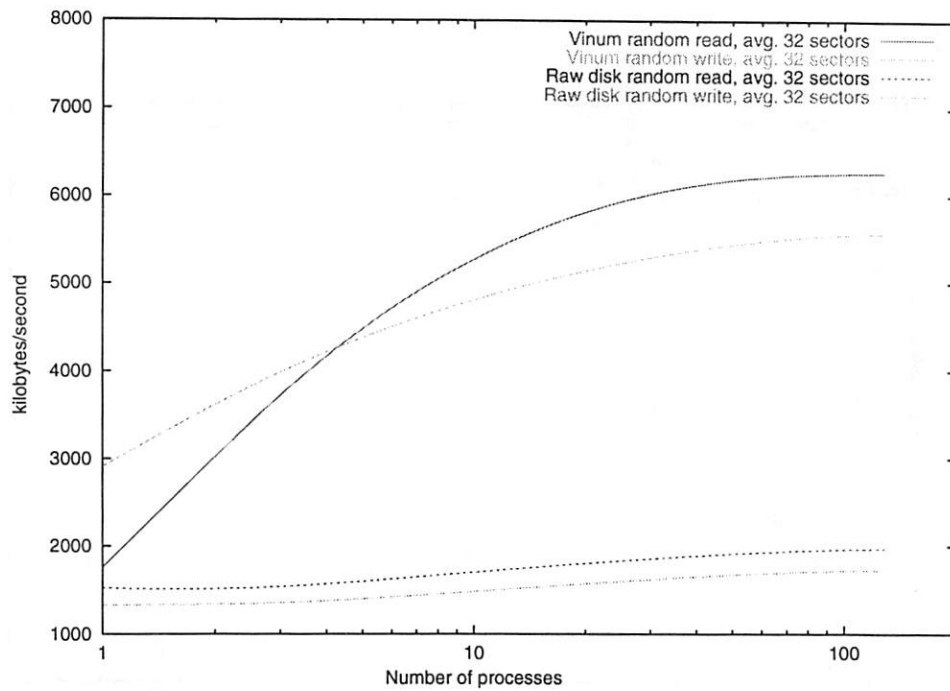


Figure 6: Concurrent random access

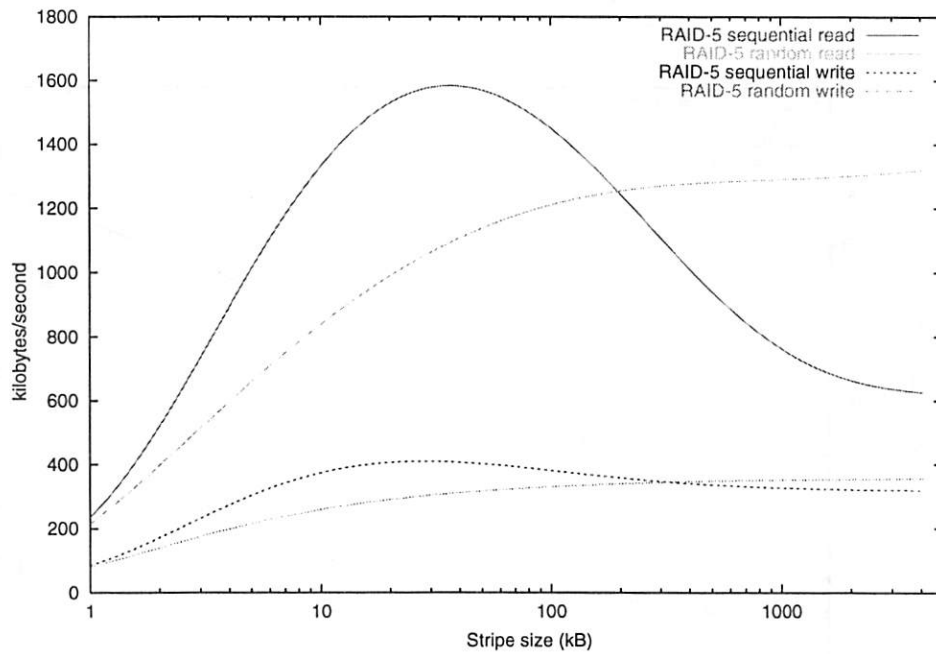


Figure 7: RAID-5 performance against stripe size

# Porting the Coda File System to Windows

Peter J. Braam  
*Carnegie Mellon University*  
Michael J. Callahan  
*The Roda Group, Inc.*  
M. Satyanarayanan  
*Carnegie Mellon University*  
Marc Schnieder  
*Carnegie Mellon University*

## Abstract

We first describe how the Coda distributed filesystem was ported to Windows 95 and 98. Coda consists of user level cache managers and servers and kernel level code for filesystem support. Severe reentrancy difficulties in the Win32 environment on this platform were overcome by extending the DJGPP DOS C compiler package with kernel level support for sockets and more flexible memory management. With this support library and kernel modules for Windows 9x filesystems in place, the Coda file system client could be ported with very little patching and will likely soon run as well on Windows 9x as on Linux. We ported Coda file servers to Windows NT. For file servers the Cygwin32 kit was used. We will not report here on the port of the Coda client to Windows NT, which is in an early stage. In both cases cross compilation from a Linux environment was most helpful to get a good development environment.

## 1. Introduction

The purpose of this paper is to convey our progress on porting a sophisticated distributed file system running on the Unix platform to Windows NT and Windows 9x. Coda [1], [10] boasts many valuable features such as read/write server replication, a persistent client cache, a good security model, access control lists, disconnected and low bandwidth operation for laptops, ability to continue operation in the presence of network and server failures and well-defined consistency semantics. Using Coda as a vehicle to study the feasibility of porting complex Unix systems to Windows should be very interesting.

The Coda project began in 1987 with the goal of building a distributed file system that had the location transparency, scalability and security characteristics of AFS [11] but offered substantially greater resilience in the face of failures of servers or network connections. As the project evolved, it became apparent that Coda's mechanisms for high availability provided an excellent base for exploring the new field of mobile computing. Coda pioneered the concept of disconnected operation and was the first distributed file system to provide this capability [13]. Coda has been the vehicle for many other original contributions including read/write server replication [1], log-based directory resolution [14], application-specific conflict resolution [15], exploitation of weak connectivity [12], isolation-only transactions, and translucent cache management [17].

Coda has been in daily use for several years now on a variety of hardware platforms running the Mach 2.6, Linux, NetBSD and FreeBSD operating systems, and most of its features are working satisfactorily. Therefore we have started to focus on further improvement of performance and ports of Coda to other platforms. Ports to NetBSD, FreeBSD and Linux have confirmed that the underlying code relies almost solely on the BSD Unix API -- as intended -- and avoids Mach specific features to the maximum possible extent. During these ports various subsystems were recognized to depend too closely on BSD specific file system data structures, and we replaced these with platform independent code, anticipating that this would greatly ease further ports in particular those to Windows. Our optimism about porting the user level code to Windows NT or Windows 95 using a POSIX implementation turned out to be justified. Our code compiled with very few patches. We will de-

scribe how this led to Coda servers running on Windows 9x and Windows NT.

However, a filesystem also depends on kernel code, and it turned out to be much less trivial to get that working on Windows 9x. We will describe these difficulties in detail. We will not report here on the kernel module for the Coda client on Windows NT.

This paper is organized by expanding briefly on the features and implementation of Coda. We then summarize some of the differences between the Win32 API and the Unix API which affect Coda. Finally we describe how we overcame serious difficulties to get clients running on Windows 9x. We finish by describing the porting effort for servers to Windows NT, and by summarizing the lessons we learned.

## 2. What is Coda?

Coda is the collective name for the programs and kernel modules which make up the Coda file servers and clients. Coda is implemented as a collection of substantial user level programs together with a small kernel module on the client which provides the necessary Coda file system interface to the operating system. The user level programs comprise Vice, the server, and Venus, the client cache manager.

The file server Vice is implemented entirely as a user-level program servicing network requests from a variety of clients. (For performance reasons, Vice can use a few custom system calls to access files by inode, but this is a detail.)

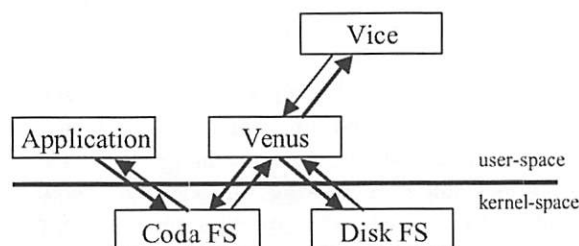


Figure 1

On the client the kernel module does some caching of names and attributes but is mostly there to re-direct system calls to Venus. The kernel module is called the *Minicache*. The user level programs make sophisticated use of the standard Unix API and they are multithreaded using a user level co-routine thread package. The filesystem metadata on clients and servers are mapped into the Vice/Venus address space and are manipulated using a lightweight transaction package called *rvm* [16].

The experience of the Coda project has been that placing complicated code in user-level processes offers tremendous development advantages without incurring unacceptable performance compromises [2]. This arrangement is shown in figure 1.

To the greatest extent possible, we wanted to replicate this structure -- and reuse code -- in the Windows ports. We faced several challenges in doing so. First, the user-level programs themselves represent a significant porting effort. Second, we must provide a kernel module to provide the Win32 file system services for Coda volumes. The second task has several components. The kernel module must translate Win32 requests to requests that Venus can service. Since the Venus interface was designed to do this for BSD Unix filesystems, and not for Win32 filesystems, some plumbing between the two filesystem models was needed. More fundamentally, given that the design uses a user level cache manager, we had to provide an environment for the user-level code that reproduces the concurrency properties of Unix upon which the Coda implementation model is based. This is discussed in more detail below. This appeared to be a highly difficult task for Windows 9x. It did not represent a fundamental problem for the Windows NT port.

For the purpose of this paper we will mostly concentrate on the client side of Coda. The two main components of a Coda client are *Venus*, the client cache manager, and the *Minicache*, which is the kernel filesystem code. When a Coda filesystem is used on a client, a program running on the client will make system calls which are directed to the filesystem in the kernel. The virtual filesystem passes this on to the *Minicache*. This does some preprocessing for the call and then passes the request on to Venus, which is running as a user-level process on the client machine. Venus contacts file servers or retrieves data from the persistent cache on the client machine and responds to the *Minicache*, after which the system call returns to the caller. In order to avoid many unnecessary context switches between the user program and the cache manager, the *Minicache* kernel code caches a small amount of naming information inside the kernel [2]. This enables the *Minicache* to complete most requests without involving Venus.

When a file is first opened, Venus resolves the name, and transfers it from a server holding the file to the client cache directory. Together with the file, the server delivers a *callback promise* to notify the client when the file has been updated by another client. This enables the client to do a subsequent opening of the file without contacting the servers, provided that callback

for the file has not been broken. Another key feature is that read and write requests are directly redirected to the so called *container file* in the cache directory on the client machine, and are not processed by the Venus. This makes read and write calls almost as efficient as they are for locally stored files. Local modifications are propagated back to the server at close time, when the Minicache contacts Venus again.

For the purpose of this paper it is important to dwell a little longer on the basic mode of operation. The requests which are passed from the kernel to Venus and vice versa are transferred through a very simple character device (especially written for Coda), we will call this the *Coda device*. The main event loop inside Venus boils down to a *select loop* on socket file descriptors for the connections with servers and the file descriptor of the Coda device. When data arrives from the Minicache, or from the servers, the select call returns to allow Venus to take appropriate action. The system call interface described above hinges on the implementation of select for the Coda device as follows. When the kernel is dispatching its system call request to Venus, on behalf of the calling process, it adds this calling process to a wait queue associated with the device. Then it wakes up Venus which proceeds to process a read system call on the Coda device, after which Venus can process the request from the kernel. When Venus is ready to deliver the result, it issues a write call to the Coda device. The write system call handling for this device finds out for which process the reply is meant, and removes this process from the wait queue. The process making the system call can now be scheduled and proceed.

### 3. Strategy

Venus contains the bulk of the code needed to get a client running, and it is quite a complicated program. The task of debugging the Minicache kernel code in the presence of such a large module is not attractive, particularly since Venus' operation depends on a correctly implemented Minicache and vice versa. The Coda team faced this before with the port of Coda to NetBSD and decided to develop *Potemkin Venus*. Potemkin is a program that appears to be a genuine Venus, connected to servers and responding to requests coming from the Minicache through the Coda device. When using Potemkin a directory tree of files residing on the "client" itself can be mounted as a Coda filesystem. Potemkin is a simple program, does no fancy caching and is merely a tool to test the implementation of the Minicache kernel

code for the Coda filesystem. It also utilizes the same select loop as Venus does.

Clearly the first element of porting Coda to Windows is to port and adapt Potemkin to Windows and to create a Minicache.

### 4. Windows 9x Problems

Although in many ways NT represents a more attractive and appropriate target for Coda, we decided to look at Windows 95 support first. Its Win32 implementation appeared to have roughly the capabilities we need (TCP/IP sockets, memory-mapped files, threads), and, unlike NT, its installable filesystem interface is documented (in the Windows 95 Device Driver Kit).

To explain the results of our first Windows 95 experiment, it helps to provide a brief description of its architecture. The Windows 95 kernel is called the Virtual Machine Manager (VMM). All Windows applications run in a single virtual machine called the System VM. Each Win32 process has its own memory context and may have multiple threads which are managed by the VMM. Other virtual machines may be created to support DOS applications; these VMs contain only a single address space and a single thread. Programs running in DOS VMs (so-called "DOS boxes") have access to the standard DOS interrupt API and a protected-mode extension called DPAPI (as well as virtualizations of typical PC hardware), but not the Win32 API. Win32 is implemented by libraries that run in user-space in the System VM and use many undocumented VMM interfaces. Fortunately, the Windows 95 DDK does specify the internal interface between the VMM component which provides file-oriented I/O to the virtual machines (called the IFSMgr) and installed filesystems. To put it in Unix terms: the (analogues of) user-level libc and kernel-internal vfs interfaces are documented, while the actual kernel system calls are largely undocumented.

Working from the DDK information, we began to implement a Potemkin-like system for Windows 95. The kernel-mode component consisted of an installable network filesystem which would enqueue file requests for a custom Win32 Potemkin venus-like process. (At this point, we made no effort to reuse Unix code, but wrote directly to the Win32 API.) The requests and responses were transferred using the Win32 DeviceIoControl API call, rather than a character device, and threads making requests synchronized with the Potemkin process by waiting on VMM semaphores.

Initially, this approach seemed to work. It was possible to start a DOS window, mount the filesystem, and manipulate files on it. However, the system would sometimes freeze if the user tried to manipulate the Potemkin drive using Windows applications. Unfortunately, this reflected a fundamental problem: the implementations of many Win32 API functions rely on 16-bit libraries in the System VM which are non-reentrant. Thread access to these libraries is serialized, system-wide, by a mutex called the Win16Mutex. The Microsoft Press book *Inside Windows 95* [3] is emphatic that the lowest-level Win32 API calls (implemented by a library called Kernel32) do not try to acquire this mutex, but unfortunately this is not so: even the very simple file I/O calls which our Win32 Potemkin made to service requests would sometimes try to acquire Win16Mutex. If the process making the request of Potemkin had already acquired it, the two processes would deadlock. Unfortunately, since the Windows user interface itself relies very heavily on 16-bit code protected by the Win16Mutex, the whole system would appear dead at this point. *Unauthorized Windows 95* by Andrew Schulman [4] and *Windows 95 System Programming Secrets* by Matt Pietrek [5] explore some of the ways that the Win16Mutex is actually used in Windows 95.

## 5. On DOS and Coda for Windows 9x

We rapidly concluded that the limitations of the Win32 API implementation in Windows 9x ruled it out as a host environment for Venus. Two possibilities at least remained. A somewhat unrealistic option would have been to implement all of Coda in the installable filesystem VMM component. A more attractive alternative was to implement the cache manager not as a Win32 process but as a DOS program. While Win32 applications often contend for the single Win16Mutex, the Windows 9x design actually gives DOS programs significantly better treatment: the actual VMM core is more reasonably flexible OS kernel which, like a Unix kernel, permits multiple VMs to have simultaneous outstanding system calls. This approach sidesteps the Win16Mutex problem and frees us from worrying about further non-reentrancy in the Windows 95 Win32 implementation. Furthermore, the application environment in a DOS box need not be as hostile as one might imagine: D.J. Delorie's DJGPP port of gcc provides a 32bit Unix-like libc, and Phar Lap sells a Win32-subset implementation that runs in DOS boxes.

We decided to explore the possibility of hosting Venus in a DOS box, using DJGPP's compiler and libc. The first

step was to identify what APIs Venus needed that were not already provided. They were:

- ♦ TCP/IP networking accessed using the standard BSD sockets API
- ♦ a *select()* call that could wait on both networking sockets and the Minicache simultaneously, and
- ♦ a limited form of *mmap()*. Specifically, Venus needs to be able to allocate virtual memory at fixed virtual addresses in its memory space. On Unix systems, this is accomplished using an anonymous *mmap()* call.

The TCP/IP implementation in Windows 9x runs in kernel mode as part of the VMM. The published Win32 networking API is implemented by a user-level library that uses undocumented VMM calls. There is no built-in support for TCP/IP access from a DOS box. However, there is a sketchily-documented internal VMM interface to the networking stack. The approach we adopted was to use this internal interface to implement a sockets-like API which could be exposed to DOS boxes.

A second, related issue was support for the *select()* call that drives the Venus event loop. This *select()* call needs to wait on multiple UDP and TCP sockets and simultaneously on the queue of requests from the Minicache. To support this, our kernel sockets module exposes an interface that the Minicache uses to participate in *select()* calls.

Finally, as mentioned above, Venus uses a package called *rvm* that provides a transactional, memory-resident database of filesystem metadata. This package relies on being able to read the database contents into the same region of virtual memory every time Venus runs. Unfortunately, DPML, the interface that allows 32-bit applications running in DOS boxes to allocate memory, does not permit the application to specify the virtual address where the newly-allocated memory block will land. (To be precise: there is a version of DPML that does specify a way to do this, but the Windows 95 implementation does not implement the feature.) As a result, we wrote a kernel module that implemented a separate memory allocation system for 32-bit DOS applications and which does permit the application to choose which virtual pages to allocate.

Once these pieces were in place, our port became surprisingly straightforward. To manage our source code effectively we resisted the temptation to go native. Instead, we cross-compile from Linux workstations. For debugging, we use gdb's remote debugging feature: a specially modified debugging stub runs Venus under debugger on Windows 95, communicating over a TCP

socket (using our socket implementation) with a gdb process running on Linux. The result is a highly effective work environment for Coda development: we can compile and debug the Windows 95 Venus without leaving xemacs!

It is perhaps worth mentioning two other aspects of our approach which significantly speeded the porting process. First, we implemented a work-around for Windows 95's lack of support of dynamically loaded filesystem drivers. We wrote a small shim filesystem driver that would load and register itself at boot time as required by the Windows 95 architecture. All actual filesystem requests it would divert to a dynamically-loaded Minicache module that we could update multiple times without rebooting. Second, during development we modified Venus to use an intermediate relay program to converse with the Minicache. That is, instead of having Venus read requests from the Minicache and send replies directly back, we had Venus read and reply to Minicache requests using a UDP socket. A separate relay program would transfer requests and replies between the Minicache and the UDP socket that Venus used. This had a couple significant advantages: a) the relay program provided an excellent debugging log of interactions between Venus and the Minicache and helped identify quickly, when a bug occurred, which component was at fault; b) by using a modified relay program, it was possible to test Venus with synthetic requests.

## 6. Coda Client on Windows 9x: Results

The experiment of porting the Coda client on Windows 95 has been gratifyingly successful. Although it is not stable enough to use in production, it is possible to install Microsoft Office into a Coda filesystem and then use the resulting installation to do real work—including using Office while disconnected from the Coda server cluster. In fact, it has even worked to install Office into a Coda filesystem while in disconnected mode and then to allow the reintegration process to update the servers with all hundreds-of-megabytes of Office code. While it is too early to report precise figures, Coda performance when in connected mode with a cluster of Linux Coda servers is within a factor of two or so of the built-in SMB client when run against Samba on similar Linux servers.

Despite the enormous differences between Unix and Windows environments Coda, with the exception of the kernel modules, builds from a single source archive, with very little conditional compilation. The overall arrangements of source are shown in figure 2.

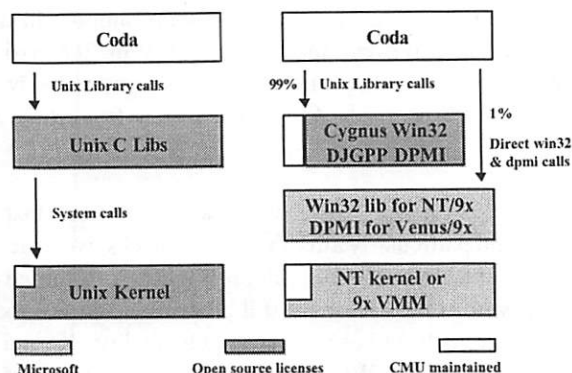


Figure 2

## 7. Coda servers on Windows 95 and NT

The Coda servers are much easier to port. They do not rely on the upcall mechanism and do not need any kernel support for their operation. Here we chose to use the Cygnus Cygwin32 library [18], which implements a Unix C library on the Win32 environment. This port proceeded smoothly and led to working servers within a few months. During the port of Coda to Linux those subsystems which appeared to rely on BSD specific data structures defined by the host environment were replaced with data structures private to the Coda package, that could exist on all platforms. This strategy proved very valuable, for porting both servers and clients, since the cache manager and file server compiled with very few patches under DJGPP and Cygwin.

One notable difficulty is that Windows NT executes asynchronous procedure calls which do not appear to mix well with Coda's user level thread library. While a work around could be found by modifying Cygnus' select call implementation, we expect that a better solution will come forward when we implement Coda threads on NT Fibers.

Initial impression were that the performance will need tuning and that some more native features are desirable, particularly on Windows NT.

## 8. Summary and lessons learned

Using a variety of freely available software packages, a very complex porting problem could be tackled. A severe lack of clear documentation on Microsoft's part, made us go wrong in several ways before finding a solution to our problems. The kernel environment for Windows 95, while difficult and hostile as a development environment, allowed the implementation of socket,

special memory allocation and filesystem support in a way that circumvented the user level Win32/Win16 mutex problems. Porting our servers was comparatively straightforward using the Cygwin32 library from Cygnus.

Looking back at our work, we vividly remember that initially and particularly after the first set backs, we were very doubtful if Coda could run on Windows. It turned out that with some creativity it did become possible to achieve our goals and obtain acceptable quality. Kernel related software for Windows is difficult to write, but the POSIX environments offered by DJGPP and Cygwin32 are truly remarkable, and should allow many other ports to proceed smoothly.

### Acknowledgements

The development of Coda over the last decade has been made possible through the generosity and sustained support of many funding agencies and corporations. The funding agencies include the National Science Foundation (NSF), under contract CCR-8657907, and the United States Air Force (USAF) and Defense Advanced Research Projects Agency (DARPA), under contract numbers F33615-90-C-1465, F19628-93-C-0193 and F19628-96-C-0061. Corporate sponsors include the Intel Corporation, and Novell Inc. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of the USAF, DARPA, Intel, Novell, CMU, or the U.S. Government.

We wish to thank members of the Coda team for discussion.

### References

1. M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E.H. Siegel, D.C. Steere, Coda: a highly available file system for a distributed workstation environment, *IEEE Transactions on Computers*, Vol. 39, No. 4, April 1990.
2. D.C. Steere, J.J. Kistler, M. Satyanarayanan, Efficient User-Level File Caching on the Sun Vnode Interface, *Proceedings of the 1990 Summer Usenix Conference*, June 1990, Anaheim, CA.
3. Adrian King, *Inside Windows 95*, Microsoft Press, 1993
4. Andrew Schulman, *Unauthorized Windows 95*, IDG Books, 1994
5. Matt Pietrek, *Windows 95 System Programming Secrets*, IDG Books, 1995
6. <http://www.microsoft.com/hwdev/ntifskit/default.htm>
7. <http://www.cygnum.com/misc/gnu-win32>
8. D.G. Korn, Porting Unix to Windows NT, Usenix 1997 Annual Technical Conference
9. <http://www.sysinternals.com/ntfilmon.htm>
10. <http://www.coda.cs.cmu.edu/index.html>
11. Howard, J.H., An Overview of the Andrew File System, Proceedings of the USENIX Winter Technical Conference Feb. 1988, Dallas, TX.
12. Mummert, L.B., Ebling, M.R., Satyanarayanan, Exploiting Weak Connectivity for Mobile File Access, *M. Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Dec. 1995, Copper Mountain Resort, CO.
13. Kistler, J.J., Satyanarayanan, M., Disconnected Operation in the Coda File System, *ACM Transactions on Computer Systems* Feb. 1992, Vol. 10, No. 1, pp. 3-25.
14. Kumar, P., Satyanarayanan, M., Log-Based Directory Resolution in the Coda File System. Proceedings of the Second International Conference on Parallel and Distributed Information Systems Jan. 1993, San Diego, CA, pp. 202-213.
15. Kumar, P., Satyanarayanan, M., Supporting Application-Specific Resolution in an Optimistically Replicated File System. Proceedings of the Fourth IEEE Workshop on Workstation Operating Systems Oct. 1993, Napa, CA, pp. 66-70.
16. Satyanarayanan, M., Mashburn, H.H., Kumar, P., Steere, D.C., Kistler, J.J., Lightweight Recoverable Virtual Memory. *ACM Transactions on Computer Systems* Feb. 1994, Vol. 12, No. 1, pp. 33-57 Corrigendum: May 1994, Vol. 12, No. 2, pp. 165-172.
17. Satyanarayanan, M., Ebling Maria E., Translucent Cache Management for Mobile Computing, to appear in proceedings of the workshop on Ubiquitous Computing, Atlanta GA, 1997.
18. Geoffrey J. Noer, Cygwin32: A Free Win32 Porting Layer for UNIX Applications, 2nd USENIX Windows NT Symposium, 1998 August 3-4, 1998 Seattle, Washington, USA

# A network file system over HTTP: remote access and modification of files and files

Oleg Kiselyov

*oleg@pobox.com oleg@computer.org oleg@acm.org*  
*http://pobox.com/~oleg/ftp/USENIX99/*

## Abstract

The goal of the present HTTPFS project is to enable access to remote files, directories, and other containers through an HTTP pipe. HTTPFS system permits retrieval, creation and modification of these resources as if they were regular files and directories on a local filesystem. The remote host can be *any* UNIX or Win9x/WinNT box that is capable of running a Perl CGI script and accessible either directly or via a web proxy or a gateway. HTTPFS runs entirely in user space. The current implementation fully supports reading as well as creating, writing, appending, and truncating of files on a remote HTTP host. HTTPFS provides an isolation level for concurrent file access stronger than the one mandated by POSIX file system semantics, closer to that of AFS. Both a programmatic interface with familiar `open()`, `read()`, `write()`, `close()`, etc. calls, and an interactive interface, via the popular Midnight Commander file browser, are provided.

## Overview

Unlike NFS and AFS, HTTP is supported on nearly all platforms, from IBM mainframes to PalmPilots and cellular phones, with a widely deployed infrastructure of proxies, gateways, and caches. It is also regularly routed through firewalls. Using standard HTTP GET, PUT, HEAD and DELETE request methods, a rudimentary network file system can be created that runs cross-platform (e.g., Linux, Solaris, HP-UX, and Windows NT) on a variety of off-the-shelf HTTP servers: Apache, Netscape, and IIS. The HTTPFS can be used either programmatically or via an interactive interface.

HTTPFS is a user-level file system, implemented by a C++ class library on a client site, and a Perl CGI script on a remote site. The C++ framework of `VNode`, `VNode_list`, `HTTPTransaction`, `MIMEDiscreteEntity` etc. classes may be employed directly. Alternatively, HTTPFS functionality can be extended to arbitrary applications by linking with a library that transparently replaces standard file system calls (e.g., `open()`, `stat()`, and `close()`). This operation does not patch the kernel or system libraries, nor does it require system administrator privileges. The interposed functions invoke the default implementations, unless a file with an "http://" prefix is accessed. The HTTPFS client framework will handle the latter case. This permits URLs being used whenever a regular file name is expected, as an argument to `open()`, `fopen()`, `fstream()`, or a command-line parameter to file utilities. No source code needs to be modified, or even recompiled.

An important feature of HTTPFS is that it can provide a file-centric view of remote resources and containers that are not necessarily files or directories on a remote computer. Anything which an HTTPFS server can apply GET, PUT, DELETE methods to, and has timestamps and size attributes, may be accessed and manipulated as if it were a file. With HTTPFS, an off-the-shelf application may `open()`, `read()`, `write()` a "file" that may in reality be a database table, an element in an XML document, a property in the registry, an ARP cache entry, or the input or output of a process.

Borrowing from database terminology, HTTPFS provides an isolation level of "Repeatable Read" for concurrent file transactions. Once a process opens a file, it will not see changes to the file made by other concurrently running processes. This isolation is different from standard POSIX semantics, which provides for a "Dirty Read" isolation – updates made to the file by other processes are visible before the file is closed. The difference in semantics is important, but only when a file is being concurrently read and modified. As was mentioned above, HTTPFS may permit a file-type access to a table of a relational database. In this particular case, the "Repeatable Read" isolation level is appropriate as it is the default for an ANSI-compliant database.

## Hypertext Transfer Protocol

HTTP is an application-level protocol for distributed, collaborative, hypermedia information systems [1]. It is a request/response protocol, where the

client submits a request to the server, the server processes the request, and sends a response to the client. HTTP is open-ended, in that it allows new request/response pairs to be defined. The message format is similar to that used by Multipurpose Internet Mail Extensions (MIME).

An HTTP transaction is in some sense a remote procedure call. An HTTP message specifies both an operation and the data on which to invoke the operation. The protocol provides facilities for exchanging data (arguments and results), and meta-data. The latter specialize a request and a response, carry authentication information and credentials, or annotate the content. Most HTTP transactions are synchronous, although HTTP/1.1 provides for asynchronous and batch modes. Furthermore, HTTP allows intermediaries (caches, proxies) to be inserted into the response-reply chain.

An HTTP request includes the name of the operation to apply and the name of the resource. Additional parameters if needed are communicated via request headers, or a request body. The request body may be an arbitrary stream of bytes. The HTTP/1.1 standard defines methods GET, HEAD, POST, PUT, DELETE, OPTIONS, and TRACE, which can be further extended by a particular server.

- The GET method retrieves the requested data along with some meta-information about the data. The data is denoted by a URI (universal resource identifier). The GET method can be conditional; if the resource has not been modified since the specified date, no data is returned. This form is useful when a cached copy of the resource exists.

- The HEAD method works similarly to the GET method, except that the server returns only the meta-data describing the properties of a resource.

- The PUT method stores the supplied data in the specified URI. Once PUT, the data will be available via a later GET.

HTTPFS maps these methods to the corresponding file access operations, while fully preserving the methods' semantics defined in the HTTP/1.1 document.

Of particular interest is the extensibility of the HTTP protocol. A client can submit arbitrary headers, which are available to the corresponding web server. The server may send arbitrary meta-data as response headers as well. In addition, a client and a server may exchange meta-information via "name=value" attribute pairs of the standard Content-Type: header.

## Implementation of HTTPFS: Client

HTTPFS is implemented by a C++ framework. It

carries out HTTP transactions with a server and maintains a local cache of fetched files and directory listings. A file being opened for reading or modification is first fetched from a server in a GET transaction. However, if the file is already in cache, a conditional GET request is issued to verify that the cached copy is up-to-date, and reload it if not. When a file is being opened for writing, an additional Pragma: header is included in the GET request to inform the server of the open mode: O\_RDWR, O\_WRONLY, O\_CREAT, O\_EXCL, O\_TRUNC or O\_APPEND. The server may then create, truncate, or lock the resource. A response from the server is translated into the result of the open() call. Reads and writes to the opened file are then directed to the local copy. On close(), if the local copy has been modified, it is written back using PUT.

Status inquiries, e.g., stat(), lstat(), readlink(), etc., are implemented by submitting a HEAD request. A Pragma: request header tells the server which particular status information about the resource is requested.

Scanning of a directory – opendir(), readdir(), closedir() – is similar to accessing a file: a GET request is issued for a directory URI, and the resulting directory listing is locally cached.

Appendix A gives a detailed mapping between the file system API and HTTP requests and responses.

## Implementation of HTTPFS: Server

A MCHFS server is one particular HTTPFS server. It is a Perl CGI script which executes HTTPFS requests and provides access to resources and containers. In the case of MCHFS, the resources and containers happen to be regular files and directories of a computer that runs this CGI script. The script thus lists directories on its own server, sends files, and accepts new content for old or newly created files.

According to a tradition, an HTTP server operates in a "chroot"ed environment. For example, when asked to retrieve a resource http://hostname/README.html, the server sends a file located at \$DOCUMENT\_ROOT/README.html (if exists), where \$DOCUMENT\_ROOT is something like /opt/apache/htdocs. MCHFS honors this convention:

```
open("http://hostname/cgi-bin/admin/MCHFS-server.pl/README.html", O_RDONLY)
```

will let you access the same \$DOCUMENT\_ROOT/README.html file. Still MCHFS offers to escape the

"chroot"ed confines and access files anywhere in its file system. This can be accomplished by using a distinguished path component `DeepestRoot`, which refers to the root of the server's file system. For example:

```
open("http://hostname/cgi-bin/admin/MCHFS-  
server.pl/DeepestRoot/etc/passwd", O_RDONLY);  
open("http://hostname/cgi-bin/admin/MCHFS-  
server.pl/DeepestRoot/WinNT/Profiles/Administ  
rator/NTusers.dat", O_RDONLY);
```

This is discussed further in the section on security considerations, below.

MCHFS allows any web browser to view directory listings and files. A directory request is returned as plain text, in a format similar to a `'ls -l'` listing. Because MCHFS understands regular GET requests, you can use a web browser to verify that MCHFS is installed and functioning properly. Any other user agent – `Wget` or the plain `telnet` – may be employed as well.

## Transparent replacement of system calls

An application accesses the file system API either using low-level `open/read/write/close` calls, or via abstract file system interfaces (e.g., standard I/O, stream I/O, or ports). The latter are implemented, under the covers, through the `open/read/write/close`. Once these low-level functions are impersonated (and extended to handle `http://` "file names"), HTTPFS becomes available to any application without modifying the application's source code.

One does not need to patch the kernel or system libraries to intercept the POSIX filesystem API calls. One can do it safely, and without system administrator privileges by linking the application with replacement versions of these low-level API functions. The recipe for doing so is as follows:

- compile a stub function with the name of the replaced routine;
  - partially and statically link the stub with default implementations of the functions being intercepted;
  - link the result with an application, the HTTPFS client library, and necessary standard libraries; the link mode may be either static, dynamic, or mixed.
- Source code for an application is not required, only its object (compiled) form; the application need not be aware that it is using HTTPFS.

A web page [2] explains this technique in detail, and discusses another use of this interception approach: implementing processes-as-files.

## HTTPFS and Midnight Commander

The Midnight Commander is a directory browser/file manager for Unix-like operating systems [3]. Its interface is similar to that of John Socha's Norton Commander for DOS as well as to Microsoft Windows' Explorer. The Midnight Commander (MC) can show the contents of two directories at the same time. Besides the file names, the views may display size, type, modification date, and other file attributes. The MC lets you select a group of files from the current view, and perform a number of operations (copy, rename, view, edit, etc) on the current file or selection with one or few keystrokes or mouse clicks.

MC supports remote file access via MCFS, a remote file access protocol that requires that an MCFS server be running on a remote machine. I provide an "adapter" – an MCFS server linked to a HTTPFS client – that translates MCFS orders to HTTPFS requests. Thus Midnight Commander gains an ability to access remote files via HTTPFS for free, without any modifications to its code.

The following sample session hopefully shows what good the HTTPFS/MC alliance can do. This is a transcript of an actual session, with only hostnames changed.

```
• mc -d mc:sol-server/sol-  
server:80/cgi-bin/admin/MCHFS-  
server.pl/DeepestRoot/tmp
```

This command launches MC and has it display the listing of a `/tmp` directory on a remote computer `sol-server` (SunSparc/Solaris 2.6). The `mc:sol-server` component in the "directory name" above refers to the computer that executes the MC/HTTPFS adapter. The adapter may run on the same computer with MC, or alongside the HTTPFS server, or on some other site.

- Select a file on the current pane, and press F3. A built-in MC viewer shows the contents of that remote file.
- With the selection bar still on that file, press F4. A built-in editor is launched, which lets you alter the remote file as if it were a local file.

- Press F5 to copy the file to a local directory listed on the other MC's pane.

```
• Switch to that pane and type  
cd mc:sol-server/winnt-server/cgi-  
bin/admin/MCHFS-server.pl/wwwroot
```

The pane lists a remote directory `DocumentRoot/wwwroot` on a WinNT host `winnt-server`, which runs IIS. The first MC pane still shows the contents of the `/tmp` directory on `sol-server`. By selecting files and pressing F5, you may

copy files from one remote directory onto the other. In this example, MC, MC/HTTPFS adapter, and HTTPFS server are running on three different computers.

- You notice a file `US98talk.tar.gz` in the `sol-server:/tmp` directory. If you highlight the file and press `F3`, you can navigate this *remote* tar archive as if it were a directory tree. You can select files (members of that remote archive), view and copy them as if they were on your local filesystem.

## Pushing the envelope and security holes

The MCHFS script obviously opens up the file system of a host computer to the entire world. Furthermore, if a particular HTTPFS server chooses to interpret GET/PUT requests as output/input from an application (`sh` in particular), the whole system becomes exposed. Clearly this may not be desirable. Therefore, one may want to restrict access to MCHFS to trusted hosts or users. These authentication/authorization policies are the responsibility of a web server's administrator; MCHFS need not be aware of them.

In addition, the MCHFS server may implement its own resource restriction policies. For example, it can refuse PUT requests, which effectively makes exported file systems read-only. MCHFS could permit modification or listing of only certain files, or disallow use of `DeepestRoot` and `".."` in file paths, thus confining users to a limited part of the file system tree.

## Related work

HTTPFS is similar to FTPFS, a virtual file system used by Midnight Commander, Emacs and KDE to access remote FTP sites. There is also a similarity to NFS. There are, however, a number of differences:

- HTTPFS operates through TCP channels using HTTP, a simple stateless reliable protocol. HTTP is less resource-hungry than FTP.
- HTTPFS can talk to *any* host that runs an HTTP server and capable of executing a Perl CGI script.
- HTTPFS works transparently through firewalls, HTTP proxies and Web caches.
- HTTPFS also stands to benefit from various caching, load-balancing and replication facilities that web gateways offer.
- HTTPFS can rely on authentication mechanisms already built into Web servers, in addition to its own access control.
- HTTPFS can serve "files" and list "directories"

that are created on the fly. In particular, HTTPFS permits browsing of a remote *database* as if it were a local filesystem.

- Whenever a remote file or directory get accessed or modified, HTTPFS can synchronously fire up triggers and run hooks. This is very difficult to accomplish with FTP.

See [4] for a description of another data-distribution service that builds upon HTTP riches. Design of a Linux-specific HTTP-based filesystem, in the context of WebDAV, users and perfs, is discussed in [5].

## Availability and installation

The MCFS/HTTPFS adapter distribution is freely available from a HTTPFS web page

<http://pobox.com/~oleg/ftp/HTTP-VFS.html>

The distribution archive contains the complete and self-contained source code for the server and the adapter, and an `INSTALL` document. A manifest file tells what all the other files are for.

I have personally run the MCHFS on HP-UX and SunSparc/Solaris with Netscape and Apache HTTP servers, and on Windows NT running IIS. The HTTPFS client – the MC/HTTPFS adapter in particular – ran on Sun/Solaris, HP-UX, and Linux platforms. The adapter successfully communicated with a Midnight Commander on a Linux host (MC version 4.1.36, as found in S.u.S.E. Linux distribution, versions 5 and 6).

I have not yet implemented the `unlink()`, `rename()`, `mkdir()`, and `chmod()` file system calls. I should also look into persistent HTTP connections and an option of transmitting only selected pieces of a requested file, which HTTP 1.1 allows (and encourages).

## Summary: the OS is the browser

This article presents a poor-man's network file system, which is simple, very portable, and requires the least privileges to set up and run. HTTPFS offers a glimpse of one of Plan9's jewels – a uniform file-centric naming of disparate resources – but without Plan9. This file system showcases HTTP, which is capable of far more than merely carrying web pages. HTTP can aspire to be the kingpin protocol that glues computing, storage, etc. resources together to form a distributed system – the role 9P plays in Plan9 [6].

The design of HTTPFS suggests that, contrary to a cliché, it is the OS that is the browser. While Active Desktop lets you view local files and directories as if

they were web pages, HTTPFS allows access to remote web pages and other resources as if they were local files. HTTPFS has all the attributes of an OS component: it implements (a broad subset of) the filesystem API; it maintains "vnodes" and "buffer caches"; it interacts with a persistent store and offers a uniform file-centric view of various remote resources. On the other hand, HTTPFS provides a superset of remote access services every Web browser has to implement on its own. The HTTPFS and other local and network filesystems manage storage and distribution of content, while an HTML formatter along with `xv`, `ghostscript` and similar applications provide interpretation and rendering of particular kinds of data. Thus as far as the OS is concerned, viewing a web page is to be thought similar to displaying an image file off an NFS-mounted disk, and searching the Web is no different than executing `find/grep` on a local filesystem.

## References

[1] "HTTP Version 1.1," R. Fielding, J. Gettys, J. Mogul, H. Frystyk Nielsen, and T. Berners-Lee, January 1997. RFC-2068

[2] "Patch-free User-level Link-time intercepting of

system calls and interposing on library functions," Oleg Kiselyov <<http://pobox.com/~oleg/ftp/syscall-interpose.html>>

[3] "The Midnight Commander"  
<<http://www.gnome.org/mc/>>

[4] "Pushing Weather Products via an HTTP pipe. Introduction to Metcast," Oleg Kiselyov  
<<http://zowie.metnet.navy.mil/~spawar/JMV-TNG/>>

[5] "An HTTP filesystem for Linux?"  
<<http://rufus.w3.org/linux/httpfs/>>

[6] "Plan 9 from Bell Labs," Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, Phil Winterbottom  
<<http://plan9.bell-labs.com/plan9/doc/9.html>>

## Acknowledgement

Comments, suggestions, and shepherding by Chris Small are greatly appreciated.

## Appendix A

Mapping between file system API and HTTP requests and responses

### File System API call

`open filename-URL oflags mode`

where `xxxx` encodes the file status flags and file access modes as given by `oflags`: `O_RDONLY`, `O_RDWR`, `O_WRONLY`, `O_CREAT`, `O_EXCL` and `O_TRUNC`. The HTTPFS server delivers the file if needed, and verifies that the resource can indeed be retrieved, modified, created or truncated. A `VNodeFile` is created to describe the opened resource and point to a local file that holds the (cached) copy of the resource. This local file is then opened, and the corresponding handle is returned to the caller.

If the file is being opened for modification, a `dirty` bit of the `VNodeFile` is set.

A `VNodeFile` corresponding to the `filename-URL` might have already existed in a `VNode` cache. In that case, the GET request will include an `If-modified-since: yyyy` header, where `yyyy` is the value of a `VNode::last_checked` field in HTTP date format.

`close cached-file-handle`

Locate a `VNode` whose opened cache file has a handle equal to the `cached-file-handle`.

If the `filename-URL` has been opened for writing (that is, `VNodeFile::dirty` is set), upload the contents of the cache file to the HTTPFS server. The `VNode` and its cached content are not immediately disposed of, but rather stay around until "garbage-collected".

`read cached-file-handle buffer count`

Perform a regular `read(2)` operation on the `cached-file-handle`.

`write cached-file-handle buffer count`

### HTTP request issued

`GET filename-URL`

`Pragma: httpfs="preopen-xxxx"`

`If-modified-since: yyyy`

`PUT filename-URL`

None

None

Perform a regular write (2) operation on the cached-file-handle.

lseek cached-file-handle offset whence None  
Perform a regular lseek (2) operation on the cached-file-handle.

stat filename-URL struct-stat-buffer HEAD filename-URL  
Pragma: https="stat"

First we check to see if there is a valid VNode for the given filename-URL (possibly with a '/' appended, in case it turns out to be a directory). If such a VNode is found, its cached status information is immediately returned and a HTTPFS server is not bothered. Otherwise, we issue the HEAD request and fill in the struct-stat-buffer from the status-info\* in a Etag: response header.

lstat filename-URL struct-stat-buffer HEAD filename-URL  
Pragma: https="lstat"

Similar to the stat API call above.

readlink filename-URL filename-buffer HEAD file-name  
Pragma: https="readlink"

Fill in the filename-buffer with the response from the server.

opendir dirname-URL GET dirname-URL  
If-modified-since: yyyy

A new VNodeDir is created for the dirname-URL, unless the corresponding valid VNodeDir happens to exist in the VNode cache. In the latter case, the GET request will carry the If-modified-since: yyyy header with yyyy being the value of a VNode::last\_checked field.

The server returns the listing of the directory: for each directory entry (including . and ..) the server writes a line  
name/status-info\*

This listing is written as it is into a cache file of the VNodeDir. The VHandle of this VNodeDir is returned as the result of the opendir() call.

readdir VNodeDir-handle None

The dir-handle is supposed to be a VHandle of a VNodeDir. This VNode is located, its cache file is parsed and sent to a MCFS client (as a sequence of name, stat-for-the-name pairs).

closedir VNodeDir-handle None

The VNodeDir-handle is supposed to be a VHandle of a VNodeDir, which is thus closed.

rmdir dirname-URL DELETE dirname-URL  
mkdir dirname-URL mode PUT dirname-URL  
unlink filename-URL DELETE filename-URL

\* status-info, the status information for a remote resource, is a string of 11 numbers separated by a single space: "dev ino mode nlink uid gid size atime mtime ctime blocks". All numbers are in decimal notation, except mode which is octal. The meaning of the numbers is the same as that of the corresponding fields in a stat structure. See also a stat entry in Perl documentation. The status-info is a "hard validator" of a resource – resource's unique identification. Indeed, should the file be altered, at least its modification timestamp will change. The status-info is delivered in a ETag: response header, a field designated by the HTTP standard to carry (unique) resource identifiers.

# A Future-Adaptable Password Scheme

Niels Provos and David Mazières

{provos,dm}@openbsd.org

*The OpenBSD Project*

## Abstract

Many authentication schemes depend on secret passwords. Unfortunately, the length and entropy of user-chosen passwords remain fixed over time. In contrast, hardware constantly improves and attackers gain increasingly much computational power. As a result, password schemes such as other UNIX user-authentication systems are failing with time.

This paper discusses ways of building systems in which password security keeps up with hardware speeds. We formalize the properties desirable in a good password system, and show that the computational cost of any secure password scheme must increase as hardware improves. We present two algorithms with adaptable cost—*eksblowfish*, a block cipher with a purposefully expensive key schedule, and *bcrypt*, a related hash function. Failing a major breakthrough in complexity theory, these algorithms should allow password-based systems to adapt to hardware improvements and remain secure well into the future.

## 1 Introduction

As microprocessors grow faster, so does the speed of cryptographic software. Fast cryptography opens many opportunities for making systems more secure. It renders encryption usable for a wide range of applications. It also permits larger values of tunable security parameters such as public key length. Increasing security parameters makes cryptography exponentially (or at least superpolynomially) more difficult to break, dwarfing any benefit faster hardware may offer attackers. Unfortunately, one security parameter—the length and entropy of user-chosen passwords—does not scale at all with computing power. While many systems require users to choose secret passwords for authentication, few ac-

tually adapt their algorithms to preserve security in the face of increasingly powerful attackers.

One widespread use of passwords, and a good example of failure to adapt, is the UNIX password system. UNIX, a multi-user operating system, requires users to prove their identity before accessing system resources. A user typically begins a session by providing her username and secret password to a login program. This program then verifies the password using a system-wide password file. Given the importance of keeping passwords secret, UNIX does not store plaintext passwords in this file. Instead, it keeps *hashes* of passwords, using a one-way function, *crypt(3)* [10], that can only be inverted by guessing preimages. To verify a password, login hashes the password and compares the result to the appropriate hash in the password file.

At the time of deployment in 1976, *crypt(3)* could hash fewer than 4 passwords per second. Since the only known way of inverting *crypt(3)* is to guess preimages, the algorithm made passwords very difficult to recover from their hashes—so much so, in fact, that the designers of UNIX left the password file readable by all users. Today, 23 years later, a fast workstation with heavily optimized software can perform over 200,000 *crypt(3)* operations per second. Attackers can now expediently discover plaintext passwords by hashing entire dictionaries of common passwords and comparing the results to entries in a password file. *crypt(3)* nonetheless still enjoys widespread use, and legacy software even forces many sites to keep their password files readable by all users.

Today we have authentication schemes considerably more sophisticated than the UNIX password file. In practice, however, implementations of these schemes still often depend on users remembering secret passwords. There are alternatives, such as issuing special authentication hardware to users or giving them printed lists of randomly generated access codes, but these approaches generally inconvenience users or

incur additional cost. Thus, passwords continue to play an important role in the vast majority of user-authentication systems.

This paper discusses ways of building systems in which password security keeps up with hardware speeds. We present two algorithms with adaptable cost—*eksblowfish*, a block cipher with a purposefully expensive key schedule, and *bcrypt*, a related hash function. Failing a major breakthrough in complexity theory, these algorithms should allow password-based systems to adapt to hardware improvements and remain secure 20 years into the future.

The rest of the paper is organized as follows: In Section 2, we discuss related work on password security. In Section 3, we explain the requirements for a good password scheme. Section 4 presents *eksblowfish*, a 64-bit block cipher that lets users tune the cost of the key schedule. Section 5 introduces the variable-cost *bcrypt* password hashing function and describes our implementation in the OpenBSD operating system. Finally, Section 6 compares *bcrypt* to two previous password hashing functions.

## 2 Related Work

Password guessing attacks can be categorized by the amount of interaction they require with an authentication system. In *on-line* attacks, the perpetrator must actually make use of an authentication system to check each guess of a password. In *off-line* attacks, an attacker obtains information—such as a password hash—that allows him to check password guesses on his own, with no further access to the system. On-line attacks are generally considerably slower than off-line ones. Systems can detect on-line attacks fairly easily and defend against them by slowing the rate of password checking. In contrast, once an attacker has obtained password verification information, the only protection a system has from off-line attacks is the computational cost of checking potential passwords.

Techniques for mitigating the threat of off-line password guessing generally aspire to one of two goals—limiting a system's susceptibility to off-line attacks or increasing their computational cost. As a simple example of the former, many modern UNIX systems now keep password hashes secret from users, storing them in a read-protected *shadow* password file

rather than in the standard openly readable one.

Much of the work on preventing off-line password attacks has centered around communication over insecure networks. Gong et. al. [7] suggest several protocol design tricks to thwart password guessing by network attackers. Unfortunately, their most interesting proposals require encryption algorithms with unusual and difficult to achieve properties.

Several people have designed secure password protocols that let users authenticate themselves over insecure networks without the need to remember or certify public keys. Bellare and Merritt [2, 3] first proposed the idea, giving several concrete protocols putatively resistant to off-line guessing attacks. Patel [12] later cryptanalyzed those protocols, but people have since continued developing and refining other ones in the same vein. More recent proposals such as SRP [17] show promise of being secure.

Of course, even a secure password protocol requires some server capable of validating users with correct passwords. An attacker who obtains that server's secret state can mount an off-line guessing attack. Since secure password protocols require public key cryptography [8], they do have a tunable key length parameter. However, this parameter primarily controls the difficulty of mounting off-line attacks without a server's secret state; it only indirectly affects the cost of an off-line attack given that state. Tuning key length to preserve password guessing costs would have other unintended consequences, for instance increasing communication bandwidth and costing servers unnecessary computation. By combining a scheme like SRP with the *bcrypt* algorithm presented in this paper, however, one can tune the cost of guessing passwords independently of most other properties of the protocol.

Whatever progress occurs in preventing off-line attacks, one can never rule them out entirely. In fact, the decision to have an openly readable password file was not an oversight on the part of the UNIX system designers [10]. Rather, it was a reaction to the difficulty of keeping the password file secret in previous systems, and to the realization that a supposedly secret password file would need to resist off-line guessing anyway. This realization remains equally true today. Aside from the obvious issues of backup tape security, attackers who compromise UNIX machines routinely make off with the list of hashed passwords, whether it was shadowed or not. A poor hashing algorithm not only complicates re-

covery from break-ins, it also endangers other machines. People often choose the same password on multiple machines. Many sites even intentionally maintain identical password files on all machines for administrative convenience. While shadow password files certainly do not hurt security, the big flaw in UNIX password security was not its openly readable password file. Rather it was the choice of a hash function that could not adapt to a 50,000 fold increase in the speed of hardware and software. This paper presents schemes that can adapt to such an improvement in efficiency.

People have already proposed numerous schemes to increase the cost of guessing passwords. The FreeBSD operating system, for instance, introduced a replacement for *crypt(3)* based on the MD5 [14] message digest algorithm. MD5 *crypt(3)* takes considerably longer to compute than the original *crypt(3)*. Unfortunately, it still has a fixed cost, and thus cannot not adapt to faster hardware. As time passes, MD5 *crypt(3)* will offer steadily decreasing protection against off-line guessing attacks. Significant optimizations have already been found to speed up calculation of MD5 *crypt(3)* on passwords less than 16 characters.

Abadi et. al. [1] propose strengthening user-chosen passwords by appending random bits to them. At authentication time, software uses the known part of the password and a hash of the full password to guess the random bits. As hardware gets faster, one can easily tune this technique by increasing the number of random bits. Unfortunately, password strengthening inherently gives unauthenticated users the ability to mount off-line guessing attacks. Thus, it cannot be used in systems that also attempt to limit the possibility off-line attacks.

Finally, many systems rely less directly on password security for authentication. The popular ssh [18] remote login program, for example, allows users to authenticate themselves using RSA encryption. Ssh servers must have a user's RSA public key, but they need not store any information with which to verify user-chosen passwords. The catch is, of course, that most users store their private keys somewhere, and this usually means on disk, encrypted with a password. Worse yet, ssh uses simple 3-DES to encrypt private keys, making the cost of guessing ssh passwords comparable to the cost of computing *crypt(3)*. Nonetheless, because of its flexibility, ssh's RSA authentication is a generally better approach than schemes more closely tied to passwords.

For example, without modifying the core protocols, ssh could easily employ the *eksblowfish* algorithm proposed in this paper to store secret keys more securely.

### 3 Design criteria for password schemes

Any algorithm that takes a user-chosen password as input should be hardened against password guessing. That means any public or long-lived output should be of minimal use in reconstructing the password. Several design criteria can help achieve this goal.

One would ideally like any password handling algorithm to be a strong one-way function of the password—That is, given the algorithm's output and other inputs, an attacker should have little chance of learning even partial information he could not already have guessed about the password. Unfortunately, one-way functions are defined asymptotically with respect to the input length. Because there is a fixed limit to the size of passwords users will tolerate, we need a different criterion.

Informally, we would like a password scheme to be “as good as the passwords users choose.” Given a probability distribution  $D$  on passwords, we define the *predictability*  $R(D)$  of the distribution to be the highest probability  $p(s)$  of any single password  $s$  in  $D$ :  $R(D) = \max_{s \in D} p(s)$ . A password handling function is secure if an attacker's probability of learning any partial information about the password is proportional to the work he invests and to the predictability of the password distribution.

More formally, let  $F(s, t)$  be a function in which the argument  $s$  represents a user's secret password. The argument  $t$  represents any other inputs to the function, which we will say are drawn from a probability distribution  $T$ . We say that  $F$  is  $\epsilon$ -secure against password guessing attacks if the following hold:

1. Finding partial information about inputs is as hard as guessing passwords. Let  $S$  be the set of all acceptable passwords (e.g.  $S = \{0, 1\}^k$ ). For any probability distribution  $D$  on  $S$ , any number  $g \in \mathbb{N}$ , and any adversary  $A$  implemented as a probabilistic circuit of  $g$  gates (and

taking  $c$  hashes as input), there exists a circuit  $M$  of equal or lesser size such that for all predicates  $P : S \rightarrow \{0, 1\}$ :

$$\begin{aligned} & \left| \Pr[t_1 \leftarrow T, \dots, t_c \leftarrow T, s \leftarrow D, \right. \\ & \quad b \leftarrow A(t, F(s, t_1), \dots, F(s, t_c)); \\ & \quad b = P(s)] \\ & \quad \left. - \Pr[s \leftarrow D, b \leftarrow M(); b = P(s)] \right| \\ & < \epsilon g \cdot R(D) \end{aligned}$$

2. Finding second preimages is as hard as guessing passwords. Let  $S$  be the set of acceptable passwords. For any distribution  $D$  on  $S$ , for any number  $g \in \mathbb{N}$  and any adversary  $A$  implemented as a probabilistic circuit of  $g$  gates:

$$\begin{aligned} & \Pr[t \leftarrow T, s \leftarrow D, s' \leftarrow A(s, t); \\ & \quad s \neq s' \wedge F(s, t) = F(s', t)] < \epsilon g \cdot R(D) \end{aligned}$$

We should first note that this definition matches our intuition about a password hashing function like *crypt(3)*. If users choose predictable enough passwords, knowing a password hash gives adversaries a large advantage—they can compare hashes of the most popular passwords to that of the password they are trying to break. If, additionally, one can guess a useful predicate without even looking at a password hash—for instance by knowing that the third character of most passwords is a lower-case letter—then clearly an adversary can guess this too. If, however, no single password occurs with particularly large probability, an adversary should need to expend a large amount of effort (as measured in circuit gates) to discover any non-trivial information about a password. Finally, we also wish to prevent an attacker from finding other strings that hash to the same value as a password; such strings may prove equivalent to passwords during authentication. The requirement of second preimage resistance guarantees such collisions are hard to find, even with knowledge of the original password.

It follows from the definition that to achieve arbitrarily small values of  $\epsilon$ , a password hashing function  $F$  must actually have a second argument,  $t^1$ . If a

<sup>1</sup>If  $F$  is a randomized algorithm, we consider the random bits to be part of  $t$  and disclose them to any adversary. When a system needs both secret random bits and user-chosen passwords in the same computation, it should first hash the password with a password function  $\epsilon$ -secure against guessing, then use the password hash as input to the randomized computation.

hash function  $F(s)$  took only a password as input, a circuit sufficiently large for a particular password distribution could invert  $F$  by looking up  $F$ 's output (or enough bits of its output) in a table of all possible passwords. The size  $g$  of a circuit implementing such a lookup table depends only on the password distribution and not on the particulars of  $F$ . As proposed by Morris and Thompson [10], however, lookup tables can be thwarted with an additional input to the hashing function, which they call a *salt*. If a random salt is chosen whenever users establish new passwords, and if the salt is large enough to ensure a negligible probability of recurrence, lookup tables offer an adversary no advantage; he may as well compute  $F$  at the time of attack.

While salted passwords avert lookup tables, given a particular salt and hash, an adversary can still mount a brute force guessing attack by evaluating  $F(s, t)$  on every possible password  $s$ . Even for the best possible  $\epsilon$ -secure password hashing function, then, the cost of evaluating  $F$  is inversely proportional to  $\epsilon$ . Usability requirements therefore effect a lower limit on  $\epsilon$ —If people can only tolerate a 1 second delay for checking passwords,  $F$  can take at most 1 second to evaluate.  $F$  should not take significantly less, however, as this would unnecessarily weaken security.

The number of gates  $g$  that an adversary can reasonably muster for attack increases constantly as hardware improves. Fortunately, so does the speed of machines that must legitimately evaluate  $F$ . That means passwords should not be hashed by a single function  $F$  with fixed computational cost, but rather by one of a family of functions with arbitrarily high cost. Rather than repeatedly throwing out functions like *crypt(3)* and MD5 *crypt(3)* to start over with more expensive but incompatible ones, systems should allow the cost of any password manipulation software to scale gracefully with a tunable parameter. Thus,  $\epsilon$  can decrease as fast as hardware improves and users will tolerate. Compromised password databases can then enjoy maximum security against off-line attacks.

Aside from salting passwords and requiring sufficient computation, one should observe several other guidelines in designing any algorithm that takes password inputs. First, make any function efficiently implementable for the setting in which it will operate. The standard *crypt(3)* algorithm is based on DES [11], a particularly inefficient algorithm to implement in software because of many

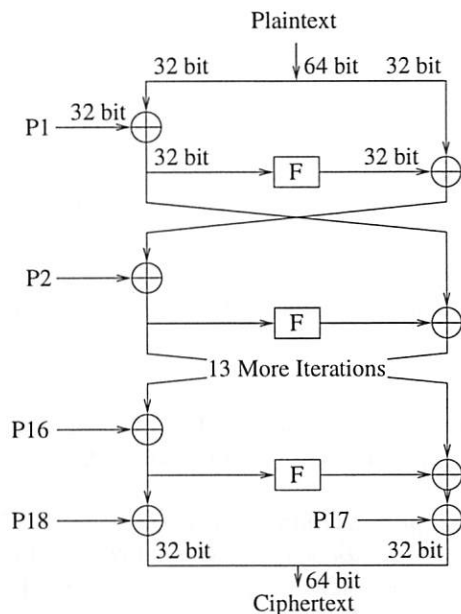


Figure 1: Blowfish Feistel network with  $F$  being the Feistel function, using only modular addition and XOR.

bit transpositions. The designers modified the algorithm enough to prevent anyone from computing *crypt(3)* using stock DES hardware. They also did not worry about custom hardware implementations. Unfortunately, Biham [4] later discovered a technique known as bitslicing that eliminates the cost of bit transpositions in computing many simultaneous DES encryptions. While, bitslicing won't help anyone log in faster, it offers a staggering speedup to brute force password searches.

In general, a password algorithm, whatever its cost, should execute with near optimal efficiency during legitimate use and offer little opportunity for speedup in other settings. It should rely heavily on fast CPUs instructions—for instance addition, bitwise exclusive-or, shifts/rotates and tables that fit in processor's first level caches. Conversely, it should avoid operations like bit transposition on which customized hardware have a large advantage. A password manipulation algorithm should not lend itself to any kind of pipelined hardware implementation. Finally, a good algorithm should derive relatively little speedup from any precomputation—for instance hashing 1,000 passwords with the same salt and hashing one password under 1,000 salts should each take nearly 1,000 times longer than hashing a single password.

## 4 EksBlowfish Algorithm

We will now describe a cost parameterizable and salted block cipher, that we call *eksblowfish* for expensive key schedule blowfish. As its base we use the blowfish block cipher by B. Schneier, that has been proven in practice and is fairly well analysed [16].

Blowfish consists of a 16-round Feistel network [15], *i.e.*, the input block is split into two halves, the right half together with a subkey is used as input for an arbitrary function  $F$ , the function result is XOR-ed with the left half, the two halves are swapped and the whole process is iterated 16 times, see Figure 1.

Blowfish uses a P-array that holds 18 32-bit subkeys derived from the encryption key. To ensure that every bit of the encryption key affects every subkey during the key schedule, Blowfish must limit encryption keys to 448 bits.

The  $F$ -function for the Feistel network uses four 32-bit S-Boxes which hold 256 entries each. A 32-bit word is split into four 8-bit parts  $a$ ,  $b$ ,  $c$  and  $d$  that are used as input for  $F$ ,

$$F(a, b, c, d) = ((S_1[a] \boxplus S_2[b]) \oplus S_3[c]) \boxplus S_4[d],$$

$a \boxplus b$  denoting modular addition and  $a \oplus b$  denoting XOR.

Figure 2 displays the key setup for *eksblowfish*. *EksBlowfishSetup* has three input parameters: the number of rounds, a salt, and the encryption key. It returns as result a key schedule. The number of rounds determine the computational cost to generate the key schedule. The salt modifies the encryption result even when the same encryption key is used. The salt does not need to be secret and may be disclosed.

At the start, *InitState()* copies the digits from  $\pi$  subsequently into the P-array and the S-boxes.

In *ExpandKey(state, 0, key)* all subkeys in the P-Boxes are XOR-ed with the encryption key. The first 32 bits of the key are XOR-ed with  $P_1$ , the following 32 bits with  $P_2$ , and so on. Subsequently, a block of zeros is encrypted using the blowfish algorithm. The resulting ciphertext replaces the subkeys  $P_1$  and  $P_2$ . An all-zero block is encrypted again, with the result replacing  $P_3$  and  $P_4$ . This process is repeated until all subkeys in the P-array and then

```

Function EksBlowfishSetup(rounds, salt, key)
1. state := InitState()
2. state := ExpandKey(state, salt, key)
3. REPEAT rounds:
    1. state := ExpandKey(state, 0, salt)
    2. state := ExpandKey(state, 0, key)
4. return state

```

Figure 2: Eksblowfish, expensive key schedule blowfish, is a cost parameterizable and salted variation of the blowfish block cipher.

all entries in the four S-boxes have been replaced.

For *ExpandKey*(*state*, *salt*, *key*) the salt instead of an infinite string of zeroes is encrypted.

After initializing, *ExpandKey* is first called with the salt and the key so that all state afterwards depend on the salt and the key right from the beginning. As result no step of the key setup can be precomputed. Following there after, *ExpandKey* is first called with the salt and then again with the key for as many iterations as the parameter *rounds* calls for.

We hope that the unpredictable and changing content of the P-array and S-Boxes may reduce the applicability of yet unknown optimizations. For example, Biham's Bitslicing optimization is possible only only after the salt and the key have been expanded into the S-Boxes, because their state changes with each iteration.

Additionally, the *eksblowfish* S-Boxes require 4 KB of memory which is constantly being accessed and modified during the key schedule. The S-Boxes cannot be shared across key schedules—separate hardware S-Boxes must exist for every simultaneous execution of the key schedule. This vastly limits the usefulness of any attempts to pipeline the Feistel network. The key setup also exhibits a very strong causal ordering with each new set of S-boxes depending heavily on the previous set. The algorithm has no obvious parallization.

## 5 Bcrypt Algorithm

The problems present in traditional UNIX password hashes led naturally to a new password scheme which we call *bcrypt*, referring to the Blowfish encryption algorithm. Bcrypt uses a 128-bit salt and

encrypts a 192-bit magic value. It takes advantage of the expensive key setup in *eksblowfish*.

The *bcrypt* algorithm runs in two phases. In the first phase, *EksBlowfishSetup* is called with the number of rounds, the salt, and the password to initialize *eksblowfish*'s state. Because of the expensive key schedule setup, most of *bcrypt*'s time is spent there, see Figure 3. Following that, the 192-bit value "OrpheanBeholderScryDoubt" is encrypted 64-times using *eksblowfish* in ECB mode with the state from the previous phase. The output is the 128-bit salt concatenated with the result of the encryption loop.

### 5.1 Implementation

In OpenBSD the 128-bit salt is generated from an arcfour (*arc4random(3)*) key stream seeded with the current time and other data generated from the kernel's entropy pool.

The kernel collects timing measurements from various devices. The data is stored in a so called entropy pool. If a userland program requests random data from the kernel, an MD5 hash is calculated over the whole entropy pool, folded in half by XOR-ing the upper and lower of the MD5 output and returned.

Using *arc4random(3)* to generate salts in a  $2^{12}$  salt space yields the same mean number of different salts for a given amount of passwords as predicted by theory.

A special configuration file, *passwd.conf(5)*, is used to determine which type of password scheme is used for a given user or group. It is possible to use different password schemes for local or YP passwords. For *bcrypt*, the number of rounds is also included. This facilitates adapting the password verification

```

1. state := EksBlowfishSetup(rounds, salt, password)
2. ctext := 'OrpheanBeholderScryDoubt'
3. REPEAT 64:
    1. ctext := EncryptECB(state, ctext)
4. RETURN Concatenate(salt, ctext)

```

Figure 3: The *bcrypt* algorithm for hashing UNIX passwords makes use of the expensive key schedule blowfish.

time to increasing processor speed. Currently, the default number of rounds for a normal user is  $2^6$ , and  $2^8$  for “root”. *bcrypt* is used in OpenBSD and has been the default password scheme since OpenBSD 2.1.

To differentiate between different password encodings each password hash starts with a unique identifier at the beginning. This allows us to use more than one password hash for `/etc/passwd`. In the current OpenBSD implementation, the identifier for *bcrypt* is “\$2a\$”.

## 6 Bcrypt Evaluation

Since password guessing is the only known method to recover plaintext passwords, we will present commonly used techniques and evaluate how they affect the security of *bcrypt*.

For further comparison with *bcrypt*, we will give a short overview of two password hashes that are in widespread use today.

### 6.1 Traditional crypt

Traditional *crypt(3)*’s design rationale dates back to 1976 [10]. It uses a password of up to eight characters as key for DES [11]. The 56-bit DES key is formed by combining the low-order 7 bits of each character in the password. If it is shorter than 8 characters the password is padded with zero bits on the right.

A 12-bit salt is used to perturb the DES algorithm, so that the same password plaintext can produce 4096 possible password encryptions, depending on the salt. A modification to the DES algorithm, swapping bits  $i$  and  $i + 24$  in the DES E-Box output

when bit  $i$  is set in the salt, achieves this and also makes DES encryption hardware useless for password guessing.

The 64-bit constant “0” is encrypted 25 times with the DES key. The final output is the 12-bit salt concatenated with the encrypted 64-bit value.

The resulting 76-bit value is recoded into 13 printable ASCII characters.

### 6.2 MD5 crypt

MD5 *crypt(3)* was written by Poul-Henning Kamp for FreeBSD. The main reason for using MD5 was to avoid problems with American export prohibitions on cryptographic products and to allow for a longer password length than the 8 characters used by DES *crypt(3)*. The password length is restricted only by MD5’s maximum message size of  $2^{64}$  bits. The salt can vary from 12 to 48 bits.

The password and salt are hashed in a number of different combinations to slow down the evaluation speed. This is done in a way which makes it doubtful that the scheme was designed from a cryptographic point of view, *e.g.*, the binary representation of the password length at some point determines which data is hashed, for every zero bit the first byte of the password and for every set bit the first byte of a previous hash computation.

The result is the concatenation of the version identifier “\$1\$”, the encoding of the salt and the 128-bit hash output, separated from the salt by another “\$”.

### 6.3 Dictionary Attacks

Dictionary attack is based on the fact that many users choose their passwords in a very predictable

n	10 digits	26 lowercase	36 lowercase alphanumeric	52 mixed case	62 mixed case alphanumeric	95 keyboard characters
4	0.04 sec	1.9 sec	7 sec	30.5 sec	61.6 sec	5.7 min
5	0.4 sec	49.5 sec	4.2 min	26.4 min	1.1 hours	9 hours
6	4.2 sec	21.5 min	2.5 hours	22.9 hours	2.7 days	35.5 days
7	41.6 sec	9.3 hours	3.8 days	49.6 days	169 days	9.2 years
8	6.9 min	10 days	136 days	7 years	28.8 years	875 years
9	1.2 hours	261 days	13.4 years	366 years	1786 years	83180 years

Table 1: Time required to test a single password against a specified password space when being able to perform 240,000 evaluations of *crypt(3)* per second. Password spaces above the separation can be searched completely within 4 days. These times are normal for traditional *crypt(3)* nowadays.

way. Often the password can be found in a dictionary or is the name of a close relative with small modifications, *e.g.*, “Susanne1” or “neme\$1\$”. The attacker compiles a list of common names and words. For a given salt, the words in the list are hashed with the password scheme and compared with entries of the same salt in the password file. If there is a match, the plaintext password has been found.

Commonly those lists contain several hundred thousands of words; dictionary attack is feasible only when the one-way function can be computed very fast.

The computation cost of *bcrypt* can be adapted to each computer system it runs on, so that dictionary attacks become in general so slow that they are rendered useless. This is not true for either *crypt(3)* or MD5 *crypt(3)*. Both of them have fixed cost and can not adapt to faster hardware.

### 6.3.1 Salt Collisions

A *salt collision* occurs when two password encodings use the same salt. Ideally, there should be no *salt collisions*, *i.e.*, the salt for each password encoding should be different even across password files. Because traditional *crypt(3)* uses only 4096 different salts, we expect many collisions as displayed in Figure 4. As a common optimization for the dictionary attack, an attacker can collect encrypted passwords with the same salt to reduce the number of computations. The resulting speedup can roughly be determined as

$$\frac{\text{number of passwords}}{\text{number of different salts}}.$$

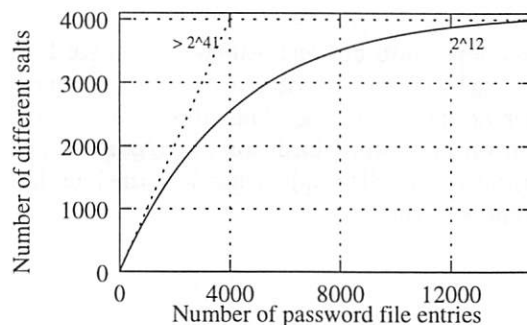


Figure 4: Distribution of expected different salts depending on the salt space against the number of entries in a password file.

If a true random number generator is used, the expected number of different salts for  $n$  password entries with  $s$  possible salts is

$$EV(n, s) = \sum_{i=0}^{n-1} \left( \frac{s-1}{s} \right)^i = s - (s-1)^n s^{1-n}.$$

For 15,000 passwords, a salt space of  $2^{41}$  ensures with a very high probability that all salts are different. For  $2^{12}$  we expect only 3991 different salts and for  $2^{24}$  about 14994. However we usually find fewer different salts than expected, because many operating systems do not generate good random numbers. We find that the salt space for traditional *crypt(3)* is obviously too small and that MD5 *crypt(3)*’s salt is sufficient when the full salt space is used. On the other hand, *bcrypt*’s 128-bit salt should guarantee that salts are unique over all time.

### 6.3.2 Precomputing Dictionaries

Precomputation means that a list of common words will be encrypted with all possible salts and stored on mass data storage. Reversing the password scheme is then a simple lookup in a database with no real computational cost.

The 1934 edition of the Webster Dictionary contains, after truncation to 8 character words and duplicate removal, 171395 unique entries. Using a 12-bit salt the result would fit on a single 9 GB hard-disk. When using QCrack [13] the whole dictionary uses only 670 MB. For each of the 4096 salts QCrack computes *crypt(3)* and hashes the result to a single byte, so that all possible encryptions of a password can be stored in 4096 bytes. When comparing an encrypted password against QCrack's database, we hash the encrypted password to one byte and compare it with the corresponding byte in the database. If they are the same we need to compute *crypt(3)*, if not we proceed with the next password in the database. Thus testing for words stored in QCrack's hash table needs only a computation of *crypt(3)* for one in 256 encrypted password entries.

Using QCrack's hashing technique about 2,350,000 words can be stored on a 9 GB harddisk.

Precomputation for traditional *crypt(3)* has lost its usefulness because the ratio of computing cost versus storage cost is in favor of just recomputing the whole dictionary each time it is needed. For other password hashes like *bcrypt*, that can not be computed as fast as traditional *crypt(3)*, the salt space has been increased, so it is questionable whether precomputation is still useful.

### 6.4 Algorithm Optimization

The attacks already mentioned determine a plaintext password and hash it using the password scheme to compare the result with an entry in a password file. Optimization allows a reduction in the computing time for a password scheme, thus making an attack more practical.

Biham recently discovered a notable optimization that replaces the DES S-Boxes with a logic gate circuit and encryption is done bit-wise [4]. Thus a 64-bit processor will be viewed as 64 parallel one-bit processors.

On a 300MHz Alpha 8400 processor, Biham found the bitsliced DES implementation to be about 5 times faster, encrypting 137 Mbps on average, than Eric Young's libdes, encrypting 28 Mbps.

For MD5 *crypt(3)* the situation is similar. In "John the Ripper" [5] a considerable speedup was made by limiting the scheme to a password length of 15, which allowed a significant simplification in the central loop of the MD5 *crypt(3)* algorithm.

### 6.5 Hardware Improvements

In 1977 on a VAX-11/780, *crypt(3)* could be evaluated about 3.6 times per second. In the last 20 years machine speed has increased dramatically and the algorithm has been optimized in various ways.

The Electronic Frontier Foundation built a DES cracker this year and was able to crack a 56-bit key in 56 hours with an average search rate of about  $88 \cdot 10^9$  keys per second [6]. Such a machine could brute force traditional *crypt(3)* in 22 days, compared to 875 years on the fastest alpha processor we had access to.

The impact of increasing processor speed and better optimization of the password hashing algorithm is shown in Figure 5.

Both traditional and MD5 *crypt(3)* operate with a fixed number of rounds. On a modern Alpha processor, traditional *crypt(3)* can already be computed fast enough to render it unusable for authentication purposes. When using specialized DES hardware the computing time can be reduced again by several orders of magnitude.

Neither traditional nor MD5 *crypt(3)* support a variable number of rounds. With increasing processing power they will become more and more easy to compute.

## 7 Conclusion

Many authentication schemes depend on secret passwords. Unfortunately, the length and entropy of the passwords users choose remain fixed over time. In contrast, hardware constantly improves

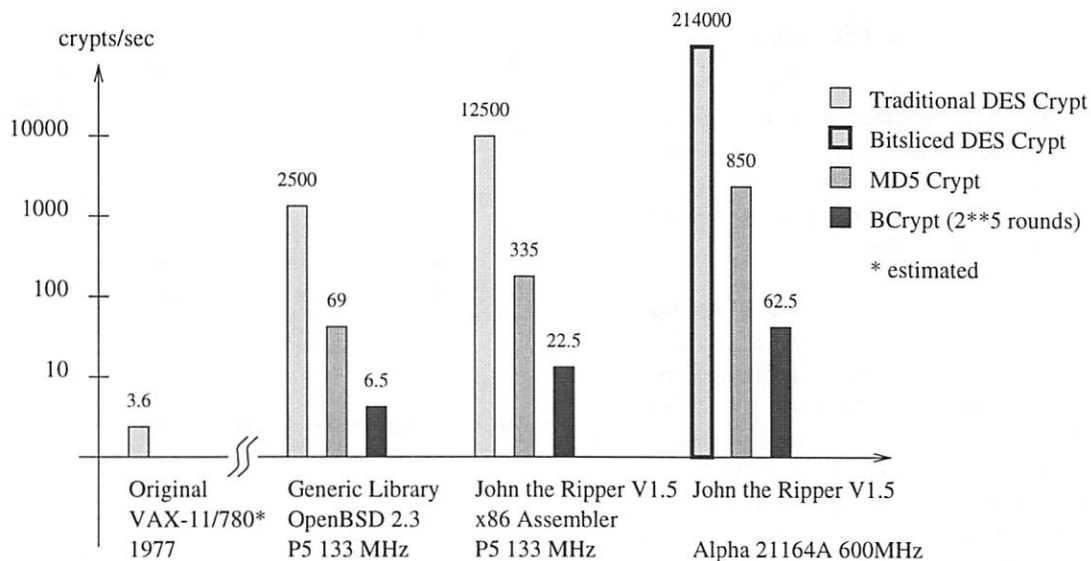


Figure 5: Impact of Algorithm Optimization and Advance in Processors

and attackers gain increasingly much computational power. As a result, password schemes including the UNIX user-authentication system are failing to hold up to off-line password guessing attacks.

In this paper, we formalize the notion of a password scheme “as good as the passwords users choose,” and show that the computational cost of such a scheme must necessarily increase with the speed of hardware. We propose two algorithms of parameterizable cost for use with passwords. *Eksblowfish*, a block cipher, lets one safely store encrypted private keys on disk. *bcrypt*, a hash function, can replace the UNIX password hashing function or serve as a front-end to secure password protocols like SRP. We have deployed *bcrypt* as part of the OpenBSD operating system’s password authentication. So far it compares favorably to the two previous hashing algorithms. No surprise optimizations have yet turned up. As hardware speeds increase, OpenBSD lets one preserve the cost of off-line password cracking by tuning a simple configuration file.

## 8 Acknowledgments

We thank Solar Designer for helpful discussions on optimization and flaws of password schemes. We further thank Angelos D. Keromytis, Peter Honeyman and Robert T. Morris for remarks and suggestions.

## References

- [1] Martín Abadi, T. Mark A. Lomas, and Roger Needham. Strengthening passwords. Technical note 1997-033, DEC Systems Research Center, September 1997.
- [2] Steven M. Bellovin and Michael Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *Proceedings of the 1992 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1992.
- [3] Steven M. Bellovin and Michael Merritt. Augmented encrypted key exchange. In *Proceedings of the First ACM Conference on Computer and Communications Security*, pages 224–250, Oakland, CA, November 1993.
- [4] Eli Biham. A Fast New DES Implementation in Software. In *Fast Software Encryption, 4th International Workshop Proceedings*, pages 260–271. Springer-Verlag, 1997.
- [5] Solar Designer. John the Ripper. <http://www.false.com/security/john>.
- [6] Electronic Frontier Foundation. *Cracking DES*. O’Reilly and Associates, 1998.
- [7] Li Gong, T. Mark A. Lomas, Roger M. Needham, and Jerome H. Saltzer. Protecting poorly chosen secrets from guessing attacks. *IEEE Journal on Selected Areas in Communications*, 11(5):648–656, June 1993.
- [8] Shai Halevi and Hugo Krawczyk. Public-key cryptography and password protocols. In *Proceedings of the 5th ACM Conference on Computer and Communications Security*, 1998.

- [9] Daniel V. Klein. “Foiling the Cracker”: A Survey of, and Improvements to, Password Security. In *Proceedings of the 2nd USENIX Security Workshop*, pages 5–14, 1990.
- [10] Robert Morris and Ken Thompson. Password Security: A Case History. *Communications of the ACM*, 22(11):594–597, November 1979.
- [11] National Bureau of Standards. Data Encryption Standard, January 1977. FIPS Publication 46.
- [12] Sarvar Patel. Number theoretic attacks on secure password schemes. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 236–247, Oakland, CA, May 1997.
- [13] QCrack.  
`ftp://chaos.infospace.com/pub/qcrack/qcrack-1.02.tar.gz`.
- [14] R. L. Rivest. The MD5 Message Digest Algorithm. RFC 1321, Apr 1992.
- [15] Michael Ruby. *Pseudorandomness and Cryptographic Applications*. Princeton Computer Science Notes, 1996.
- [16] Bruce Schneier. Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish). In *Fast Software Encryption, Cambridge Security Workshop Proceedings*, pages 191–204. Springer-Verlag, December 1993.
- [17] Thomas Wu. The secure remote password protocol. In *Proceedings of the 1998 Internet Society Network and Distributed System Security Symposium*, pages 97–111, San Diego, CA, March 1998.
- [18] Tatu Ylönen. SSH – secure login connections over the Internet. In *Proceedings of the 6th USENIX Security Symposium*, pages 37–42, July 1996.



# Cryptography in OpenBSD: An Overview

Theo de Raadt, Niklas Hallqvist, Artur Grabowski, Angelos D. Keromytis, Niels Provos  
{deraadt,niklas,art,angelos,provos}@openbsd.org  
*The OpenBSD Project*

## Abstract

Cryptographic mechanisms are an important security component of an operating system in securing the system itself and its communication paths. Indeed, in many situations, cryptography is the only tool that can solve a particular problem, *e.g.*, network-level security. While cryptography by itself does not guarantee security, when applied correctly, it can significantly improve overall security. Since one of the main foci of the OpenBSD system is security, various cryptographic mechanisms are employed in a number of different roles.

This paper gives an overview of the cryptography employed in OpenBSD. We discuss the various components (IPsec, SSL libraries, stronger password encryption, Kerberos IV, random number generators, *etc.*), their role in system security, and their interactions with the rest of the system (and, where applicable, the network).

## 1 Introduction

An important aspect of security in a modern operating system is cryptographic services and mechanisms. While not a security panacea, cryptography is sometimes the right tool in solving certain problems. In particular, cryptography is extremely useful in solving a number of security issues in the following three areas:

- Network security.
- Secure storage facilities.
- (Pseudo-) Random number generators.

Since one of our goals in the OpenBSD project is to provide strong security, we have implemented a number of protocols and services in the base system. An OpenBSD distribution thus has full support for such mechanisms as IPsec, SSL, Kerberos, *etc.*, being unaffected by export restriction laws.

Simply supporting these mechanisms, however, is not sufficient for wide-spread use. We are constantly

trying to make their use as easy and, where possible, transparent to the end user. Thus, more work is done in those mechanisms that can be used to provide transparent security, *e.g.*, IPsec.

With this paper, we intend to give a good overview of the cryptography currently distributed and used in OpenBSD, and of our plans for future work. We hope this will be of interest both to end-users and administrators looking for better ways to protect their host and networks, and to developers in other systems (free or otherwise) that are considering supporting some of these mechanisms. We should again caution the readers, however, that cryptography does not solve all security problems in an operating system, and should not be considered as an end in itself, but rather as an important piece of the security puzzle.

### 1.1 Paper Organization

The remainder of this paper is organized as follows: section 2 describes the various network security facilities implemented and supported in OpenBSD, section 3 covers the extensive use of random number generators, and section 4 briefly outlines our future plans in this area. Section 5 concludes the paper.

## 2 Communications Security

In an increasingly networked environment, communications security support in an OS is extremely important. As there are different mechanisms and different layers where one may apply security, OpenBSD supports a number of security protocols and mechanisms, some of which were developed (or even designed) by our developers. In some cases, there is considerable overlap in functionality. One of our goals is to eventually make it transparent to the end user which such security mechanism is in use.

The following sections give a brief overview of these mechanisms, some detail of their implemen-

tation and integration in OpenBSD, and our plans for future work. As we already mentioned in section 1, we consider IPsec an extremely important tool in network security, both because of its potential for user-transparency and its flexibility. This is reflected by the more thorough coverage of IPsec in the text that follows.

Other popular mechanisms, such as SSH [38], are not covered because they are only part of our ports system. While virtually all the developers use SSH, there is no free implementation we can add to our standard distribution. Furthermore, the current version of SSH is restricted by the RSA patent in the US. We are waiting for a free implementation to become available as part of the IETF standardization process of SSH. Such an implementation would be linked with our `libssl`.

## 2.1 SSL

In OpenBSD `libssl` provides a toolkit for the Secure Socket Layer (SSL v2/v3) and Transport Layer Security (TLS v1) [6] which provide strong cryptographic protection for network communication such as server authentication and data encryption. The Secure Socket Layer is currently used by web servers, *e.g.*, Apache as shipped with OpenBSD, and browsers like Netscape Communicator. In the future, applications like *telnet* and *ftp* will be converted to use TLS, possibly even during our network installation process.

Due to patent restrictions, `libssl` in the OpenBSD distribution supports only digital signatures with DSA [27], but an additional package is provided for users outside the USA to add back RSA-signature [19] support. This is implemented by providing two shared libraries: `libssl.so.1.0` has only function stubs for RSA support, while `libssl.so.1.1` contains full RSA support. Notice that shared library minor-version number changes typically indicate interface-transparent bug fixes.

## 2.2 IP Security (IPsec)

### 2.2.1 Background

While IP has proven to be an efficient and robust protocol when it comes to actually getting data across the Internet, it does not inherently provide any protection of that data. There are no facilities to provide confidentiality, or to ensure the integrity or authenticity of IP [31] datagrams. In order to remedy the security weaknesses of IP, a pair of protocols collectively called IP Security, or IPsec [3, 16] for short, has been standardized by the IETF.

The protocols are ESP (Encapsulating Security Payload) [2, 15] and AH (Authentication Header) [1, 14]. Both provide integrity, authenticity, and replay protection, while ESP adds confidentiality to the picture. IPsec can also be made to protect IP datagrams for other hosts. The IPsec endpoints in this arrangement thereby become security gateways and take part in a virtual private network (VPN) where ordinary IP packets are tunneled inside IPsec [36].

Network-layer security has a number of very important advantages over security at other layers of the protocol stack. Network-layer protocols are generally hidden from applications, which can therefore automatically and transparently take advantage of whatever network-layer encryption services that host provides. Most importantly, network-layer protocols offer a remarkable flexibility not available at higher or lower layers. They can provide security on an end-to-end (securing the data between two hosts), route-to-route (securing data passing over a particular set of links), edge-to-edge (securing data as it passes from a “secure” network to an “insecure” one), or a combination of these.

### 2.2.2 Operation

Central to both ESP and AH are an abstraction called security association, or SA. In each SA there is information (algorithm IDs, keys, *etc.*) stored describing how the wanted protection should be setup. For two peers to be able to communicate they need matching SAs at each end. When deciding what SA should be used for outbound traffic, some kind of security policy database needs to be consulted. In OpenBSD, this is currently implemented as an extension to the routing table, where source/destination addresses, protocol, and ports serve as selectors.

Looking at the wire format, IPsec works by inserting an extra header between the IP header and the payload. This header holds IPsec-specific data, such as an anti-replay sequence number, cryptographic synchronization data, and integrity check values. If the security protocol in use is ESP, a cryptographic transform is applied to the payload in-place, effectively hiding the data. As an example, an UDP datagram protected by ESP is shown in figure 1.

This mode of operation is called transport mode, as opposed to tunnel mode which is typically used when a security gateway is protecting datagrams for other hosts. Tunnel mode differs from transport mode by the addition of a new, outer, IP header consisting of the security gateways’ addresses instead of the actual source and destination, as shown in figure

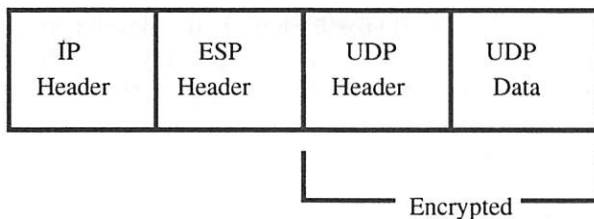


Figure 1: IPsec Transport Mode

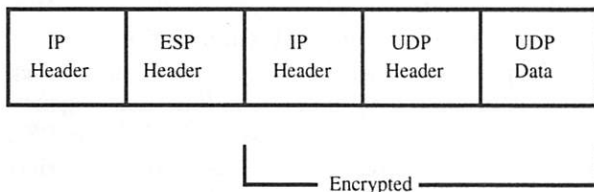


Figure 2: IPsec Tunnel Mode

2.

As was mentioned earlier, this mode is ideal for implementing VPNs.

The last, but not least, part of the picture is a key management infrastructure. IPsec can only work if the keys in the SAs are synchronized and updated in a secure fashion. To automate this task, different protocols have been devised that allow two peers to compute identical keys without actually sending all the data needed for it over the wire [7, 8]. The Internet Key Exchange, IKE, is one such, and Photuris is another. The main difference between these two lies in the complexity level. IKE is a very complex protocol which, however, offers considerable flexibility in negotiating and establishing SAs. IKE is the official IETF standard. Both protocols work in a similar vein, by first building an encrypted application-level “tunnel” where further key exchanges take place. The Diffie-Hellman algorithm [7] is used to make it computationally hard to crack the key computation. Every SA is assigned a lifetime, either in wall-clock time or in volume, and when such a lifetime expires, the key management daemon renegotiates with the peer, creating new SAs with fresh keys.

### 2.2.3 OpenBSD IPsec

OpenBSD’s IPsec stack was written by John Ioannidis and Angelos Keromytis [18] and later enhancements and fixes have been provided by Niels Provos and Niklas Hallqvist. The core is stable and in production use securing data in many places all over the world, as it does not suffer from US export regulations. A number of companies, agencies, institutions, and individuals are using the code, a fact

that has helped us significantly in finding and fixing bugs, and in motivating further development.

Recently, the API used to setup and maintain the SA database was switched to the standard PF\_KEY [23]. This API is much more flexible than the old PF\_ENCAP interface. Available algorithms for encryption are DES [26], 3DES, Cast-128, Blowfish [35], and Skipjack (support for the latter, despite its known weaknesses, was added after requests by US Government agencies using our IPsec stack). One-way hash algorithms are MD5, SHA1 and RIPEMD160 [20, 21, 17]. For key management, two daemons are available, *isakmpd* implementing IKE [29, 22, 12] and *photurisd* implementing Photuris [13].

### 2.2.4 Future Work in IPsec

Our IPsec implementation is under constant development and improvement, as there remain a number of unresolved issues.

- Our IPv6 stack is not yet integrated with our IPsec implementation.
- We want a more flexible, possibly unified policy mechanism. In particular, we are looking into merging routing, security policy, and protocol block lookups.
- Develop or borrow a policy API, rather than use private extensions to PF\_KEY and PF\_ROUTE.
- *isakmpd* has not yet covered all mandatory requirements in the RFCs.
- A DNSSEC [9] implementation, and integration in *isakmpd* and *photurisd*, will be needed for opportunistic encryption.
- *isakmpd* and *photurisd* are not linked with *libssl* so they will not automatically support RSA when an RSA-supporting *libssl* is installed.
- We do not currently do on-demand keying (a facility available in the past through the PF\_ENCAP API).
- Finally, we intend to support some application API for requesting security and possibly other services. With that in place, we intend to have all networking applications take advantage of IPsec.

All of these are improvements that we want to address in the time-frame for the next release.

## 2.3 Kerberos

In a networked environment, it is very important to be able to authenticate users in a secure way over insecure networks. Kerberos is a network authentication protocol using a trusted third-party to provide authentication and basic session-key exchange.

Kerberos is built around a central key distribution center (KDC) which keeps a database of clients and servers (called principals) and their private keys. Encryption in Kerberos is based on DES [26]. When the client wants to use some service it issues a request to the KDC for a ticket for that service. The server returns a message encrypted with the client's private key, containing three parts: a session key that can be used for encryption between the client and the server, a timestamp, and a ticket. The ticket is encrypted with the private key of the server and contains the name of the client, a timestamp, the client's network address, lifetime of the ticket, and the same session key that the client obtained. The ticket can be passed to the server for authentication.

Kerberos [24] was originally developed by project Athena at MIT, but was not exportable from the US due to legal restrictions. The cryptographic functionality was removed and a "Bones" distribution was created and exported. The cryptographic interfaces were added back by Eric Young, and KTH (The Royal Institute of Technology in Stockholm, Sweden) maintained the code outside the USA. The Kerberos implementation in OpenBSD is "kth-krb", protocol version 4, and is used in a number of utilities.

### 2.3.1 Practical Uses

The simplest use of Kerberos is to authenticate users locally on a workstation. The *login*, *xdm*, and *su* programs in OpenBSD have the necessary code to allow Kerberos authentication. The next step is to provide authentication for network protocols. The *rlogin*, *rsh*, and *telnet* programs have been modified to use Kerberos. In addition to that, they can use the session key, obtained in the authentication phase, to encrypt the data-stream for privacy. Another very practical use is in "kx" - a protocol to authenticate and forward X11 connections in a secure way. Other programs using Kerberos for authentication include *cvs*, *sudo*, and *zlock*. Kerberos authentication is also used in *AFS*.

One of our future goals is to allow kerberized applications to use IPsec services when possible, thus avoiding double-encryption (and consequently degraded performance). Furthermore, we intend to

integrate the Kerberos 5 clone being developed at KTH as soon as it is stable, especially since Kerberos IV only supports DES [26] encryption.

## 2.4 S/Key

S/Key [11, 10] is a one-time password system used for authentication. It provides protection against replay attacks where a third party captured a password, *e.g.*, by means of network sniffing, and tries to reuse it in a new authentication session.

S/Key uses a user supplied secret pass-phrase which is processed by a one-way function to generate a sequence of one-time passwords. In OpenBSD the one-way function can be chosen from a variety of computationally non-invertible hash functions like MD5 [34] or SHA1 [28], available in *libc*. S/Key is still useful when other cryptographic protocols are not available, or their implementations are not fully trusted, *e.g.*, when using a conference terminal room to login to a home machine.

## 3 Pseudo Random Number Generators

A Pseudo Random Number Generator (PRNG) provides applications with a stream of numbers which have certain important properties for system security:

- It should be impossible for an outsider to predict the output of the random number generator even with knowledge of previous output.
- The generated numbers should not have repeating patterns which means the PRNG should have a very long cycle length.
- A PRNG is normally just an algorithm where the same initial starting values will yield the same sequence of outputs.

Some applications have criteria which affect the type of PRNG which is needed. For instance, later on we will discuss IP datagram IDs and DNS [30] query-IDs, both of these issues have qualities which make it extremely desirable to have a PRNG which makes efforts to avoid emitting repetitions (thus ruling out use of a true-random source).

Many other operating systems also have random number device drivers and other related mechanisms, but largely make no use of them. Some such systems even provide such support only as optional

device drivers, therefore discouraging use (*i.e.*, reliance). OpenBSD deviates by actually using these mechanisms in numerous ways. A few major interfaces or techniques are used:

- `/dev/?random` and similar kernel interfaces
- `arc4random(3)` in `libc`
- non-repeating PRNG

Each of these, and their uses in OpenBSD, will be covered in the following sections.

### 3.1 Kernel Randomness Pool

Computers are (generally) deterministic devices making it very hard to produce real random numbers. The PRNGs we use in OpenBSD do not generate random numbers themselves. Rather, they expand the randomness they are given as input. Fortunately, a multi-user operating system has many external events from which it can derive some randomness. In OpenBSD the kernel collects measurements from various devices such as the inter-keypress timing from terminals, the arrival time of network packets, and the finishing time of disk requests. The randomness from these sources is mixed into the kernel's *entropy pool*. When a userland program requests random data from the kernel, an MD5 hash is calculated over the whole entropy pool, "folded" in half by XOR-ing the upper and lower word of the MD5 output, and returned. The user can choose the quality of the generated random numbers by reading output from the different `/dev/?random` devices.

### 3.2 `arc4random(3)`

The `arc4random(3)` interface, available in the OpenBSD `libc`, makes use of the kernel randomness pool, described in the previous section, for seeding the keystream generator employed by the *ARC4* cipher (a cipher equivalent to RSADSI's RC4). The interface provides support for applications to "add" randomness to the pool maintained by `arc4random(3)`. This interface is intended as a drop-in replacement for the traditional Unix `random(3)` interface, for those applications that need higher-quality random numbers.

### 3.3 Non-repeating Random Numbers

In OpenBSD, we designed a non-repeating pseudo-random number generator that was very fast and did not require additional resources.

For 16-bit non-repeating numbers, we used a prime  $2^{14} < p < 2^{15}$  and  $g$  a randomly chosen generator for  $\mathbb{Z}_p$ . Being a generator,  $g$  has the property that any value  $0 < x < p$  can be generated as  $x = g^y \pmod{p}$ , for some value  $y$ .

We then pick random  $a$ ,  $b$  and  $m$  with  $2^{14} < m < 2^{15}$  so that

$$f(n) \equiv a \cdot f(n-1) + b \pmod{m}$$

becomes a linear congruential generator (LCG).

We then determine the actual ID as

$$ID(n) = w \oplus (g^{f(n)} \pmod{p}),$$

where  $w$  is a random seed. After the linear congruential generator has been exhausted, the most significant bit in  $ID(n)$  is toggled and all parameters  $g$ ,  $a$ ,  $b$ ,  $m$ , and  $w$  from above are chosen anew. Because the linear congruential generator does not repeat itself and a new number space is chosen after reinitialization, the generated IDs do not repeat themselves. The PRNG is typically seeded with material from the kernel randomness pool.

#### 3.3.1 Randomness Used Inside the Kernel

- Dynamic `sin_port` allocation in `bind(2)`.

When an `AF_INET` socket is bound to a specific port number using the `bind(2)` system call, the process can choose the specific port, or elect that the system choose. Normal UNIX behavior resulted in the system allocating port numbers starting at 1024 and incrementing. Our new code chooses a random port, in the range 1024 to 49151.

A similar issue existed with reserved port creation, using the `bindresvport(3)` and `rresvport(3)` library routines, which are supposed to pick a free port in the reserved range (typically between 600 and 1023). The old behavior was to allocate decreasing port numbers starting at 1023. The old code for these library routines effected this downward search using successive calls to `bind(2)`; we have replaced this with code using a newer kernel interface which is much more efficient and chooses a random port number within the reserved range.

There are a number of poorly designed protocols (*e.g.*, `rsh`, `ftp`) which are affected by predictable port allocation; we believe that our approach is making it harder for attackers to gain an edge.

- Process PIDs.

```
char buf[20];

sprintf(buf, "/tmp/foo-%d", getpid());
(void) mkdir(buf, 0600);
```

Figure 3: The Wrong Way To Generate A "Random" Directory

Programmers often use this value as if it is random, possibly because of the compellingly attractive argument that "pid numbers are effectively random on a busy enough system." Code like "srandom(getpid())" is quite common, as is code similar to that shown in figure 3.

In a normal system the attacker will have a very easy time predicting the PID and thus the obvious race attack is trivial. The race is as follows: the attacker creates the directory first, choosing the mode and ownership; subsequently it is possible to look at and replace files in the directory.

In OpenBSD, we use randomized PIDs, with a couple of obvious exceptions, *e.g.*, *init(8)*.

- RPC transaction IDs (XID).

Sun Microsystems Remote Procedure Call (RPC) messages contain a Transaction Identifier (XID) which matches a sent query against its received reply. In most RPC systems, the XID of the first message a process transmits will be initialized using the code shown in figure 4.

Subsequently, the XID for each packet is simply incremented from this. Previously we mentioned that a local user might be able to guess what kind of range the next PID on the system might fall into; here we see that an outside attacker might also be able to determine this information. Our new code uses *arc4random()* to initialize the XID, and also avoids using two identical numbers consecutively.

- NFS RPC transaction IDs (XID).

The NFS protocol uses RPC packets for communication. The RPC XID issue also applied to the NFS code we encountered, and we now use the same mechanism for NFS XIDs.

- Inode generation numbers.

The *fsirand(8)* program makes use of *arc4random(3)* to generate random inode numbers for filesystem objects (files, directories, *etc.*). This increases the security of

```
struct timeval now;

gettimeofday(&now);
call_msg.rm_xid = getpid() ^ now.tv_sec ^
                    now.tv_usec;
```

Figure 4: Typical RPC Initialization Code

NFS-exported filesystems by making it difficult for an attacker to guess filehandles (which are partially derived from inode numbers).

- IP datagram IDs.

Each IP packet contains a 16-bit identifier which is used, if the packet has been fragmented, for correctly performing reassembly at the final destination. Previously, this identifier simply incremented every time a new packet was sent out. By looking at the identifier in a sequence of packets, an outsider can determine how busy the target machine is. Another issue was avoiding disclosure of information when using IPsec in tunneling mode, as per section 2.2.2. A naive implementation might create a new IP header with an ID one more than the ID in the existing IP header. This could lead to known-plaintext attacks [4] against IPsec.

To avoid these problems, we use the non-repeating PRNG described in section 3.3.

- Randomness added to the TCP ISS value for protection against spoofing attacks.

Inside the kernel, a 32-bit variable called *tcp\_iss* declares the Initial Send Sequence Number (ISS) to use on the next TCP [32] session. The predictability of TCP ISS values has been known to be a security problem for many years [25]. Typical systems added either 32K, 64K, or 128K to that value at various different times. Instead, our new algorithm adds a fixed amount plus a random amount, significantly decreasing the chances of an attacker guessing the value and thus being able to spoof connection contents.

- Random data-block padding for cryptographic transforms, as in RFC1827 IPsec ESP [2].

### 3.3.2 Randomness Used in Userland Libraries

- DNS query IDs typically start at 1 and increment for each subsequent query. An attacker

can cause a DNS lookup, *e.g.*, by telnetting to the target host, and spoof the reply, since the content of the query and the ID are known or easily predictable. Since host authentication is still in wide-spread use, this is a serious security vulnerability present in virtually all systems. To avoid this issue, we have modified our in-tree copy of *bind(8)* and our *libc* resolver to make use of the non-repeating PRNG.

- *arc4random(3)* seeding, as mentioned in section 3.2.
- Stronger temporary names.

Processes typically create temporary files by generating a random filename via *mktemp(3)* and then opening that file in the */tmp* directory. A more secure way for doing so is through *mkstemp(3)*, which generates the filename and opens the file in one atomic operation, thus eliminating the potential for races. Both functions, which reside in *libc*, make use of *arc4random(3)* to generate the random filenames, making it much harder for an attacker to guess the names in advance.

- Generate salts for the various password algorithms. For some more details, see section 4.1.

### 3.3.3 Randomness Used in Userland Programs

- For generating fake S/Key challenges.

One problem with most versions of RFC1938-based one time password (OTP) systems is that it is often possible to use them to determine whether or not a user has an account on a machine. The most trivial example of this is systems that provide a different prompt if the user has an entry in the OTP database. However, even for systems that always provide an OTP prompt, the prompt itself is rarely convincing and can be trivially identified as a fake. To address this problem, the OTP code in OpenBSD generates a consistent, credible challenge for non-existent users and users without an entry in the OTP database. It does so by generating the prompt based on the hostname and a hash of the username and the contents of a file generated from the kernel random pool. This file is usually created at install time and provides a constant source of random data. Thus, all three components of the challenge are constant, but only the hostname and username are known to the attacker.

- *isakmpd* and *photurisd* use the kernel randomness pool for generating IKE “exchange identifiers” (*i.e.*, protocol cookies and message IDs), random Diffie-Hellman [7] values, and random nonces.
- Certain games make use of the *arc4random(3)* interface for higher quality random numbers.

## 4 Secure Storage

One of the areas of least development in OpenBSD has been that of secure storage. While a number of utilities (*e.g.*, *vi(1)*, *ed(1)*, *bdes(1)*, *etc.*) directly support encryption services, our goal is to provide this service as transparently as possible to users. Ideally, we would like a layer either over or under the current native filesystem that would provide safe storage services.

As an interim solution, CFS [5] is included in the OpenBSD ports system and can be readily used. However, it does not provide the level of transparency we would like, and its performance is well below what we consider acceptable for general use. Clearly, more work is needed in this area.

Another issue related to secure storage is that of secure logging. Logs (and especially security-related logs) are extremely important in determining whether a system is under attack or has been compromised. The current logging facility, *syslog*, does not provide any facilities for detecting log-tampering, other than the option to send log messages to another host's *syslogd*. We are currently porting the *ssyslog* package [37] and are hoping to seamlessly replace the currently-used *syslogd*.

The remainder of this section briefly covers our *bcrypt*, approach to protecting user passwords, developed inside OpenBSD.

### 4.1 Bcrypt

Increasing computational power makes the use of cryptography to further system security more feasible and allows for more tuneable security parameters such as public key length. However, one security parameter - the length and entropy of user-chosen passwords - does not scale at all with computing power. Many systems still require user-chosen secret passwords which are hashed to keep them secret. When the UNIX password hash *crypt(3)* was introduced in 1976, it could not hash more than four passwords per second. With increasingly more powerful attackers it is common to compute

more than 200,000 password hashes per second. In OpenBSD we use the *bcrypt* algorithm to make the cost of password hashing parameterizable. Its design makes it hard to optimize *bcrypt*'s execution speed or use commodity hardware instead of software. *bcrypt* uses a 128-bit salt and encrypts a 192-bit magic value. It takes advantage of the fact that the Blowfish algorithm (used in the core of *bcrypt* for password hashing) needs a fairly expensive key setup, thus considerably slowing down dictionary-based attacks. *bcrypt* uses the *arc4random(3)* interface for password salt-generation. A comparison between this approach and the mechanism used in certain other Unix systems for generating salts has shown that while *arc4random(3)* behaved extremely close to the statistical theoretical expectations; in contrast, other systems produced large numbers of collisions, making dictionary attacks faster.

A special configuration file, *passwd.conf(5)*, is used to determine which type of password scheme is used for a given user or group. It is possible to use different password schemes for local or YP passwords. For *bcrypt*, the number of rounds is also included. This facilitates adapting the password verification time to increasing processor speed. Currently, the default number of rounds for a normal user is  $2^6$ , and  $2^8$  for "root." *bcrypt* is used in OpenBSD as the default password scheme since version 2.1. For more details, see [33].

## 5 Conclusion

In this paper, we gave an overview of the cryptography used in OpenBSD. We presented the supported network security mechanisms, with particular emphasis on IP security. We then discussed the various uses of randomness throughout the system. Finally, we briefly covered our plans for future work in the area of secure storage.

A lot of work remains to be done. In the short term, we need to complete the remaining parts of those mechanisms still under development, keeping in mind of course that security (and standards) is a moving target, and constant maintenance and updating will be needed. Beyond that, integration with existing and new utilities is a major item in our agenda. Finally, we are considering new mechanisms that address different problems, *e.g.*, untrusted-code containment.

It is important to note that all the mechanisms described in this paper are currently in use, solving real problems. We hope that this paper will encourage others to add these or similar mechanisms in

their systems.

## 6 Acknowledgments

We would like to thank Hugh Graham, Todd Miller, and Chris Turan who provided comments (and sometimes text) in earlier versions of this paper. We would also like to thank all the OpenBSD developers for the work they contribute to the project, and our users for their continuing support.

## 7 Availability

All the software described in the paper is available through the OpenBSD web page at

<http://www.openbsd.org/>

## 8 Disclaimer

OpenBSD is based in Calgary, Canada. All individuals doing cryptography-related work do so outside countries that have limiting laws.

## References

- [1] R. Atkinson. IP Authentication Header. RFC 1826, August 1995.
- [2] R. Atkinson. IP Encapsulating Security Payload. RFC 1827, August 1995.
- [3] R. Atkinson. Security Architecture for the Internet Protocol. RFC 1825, August 1995.
- [4] S. Bellovin. Probable Plaintext Cryptanalysis of the IP Security Protocols. In *Proceedings of the Symposium on Network and Distributed System Security*, pages 155–160, February 1997.
- [5] M. Blaze. A Cryptographic File System for Unix. In *Proc. of the 1st ACM Conference on Computer and Communications Security*, November 1993.
- [6] T. Dierks and C. Allen. The TLS protocol version 1.0. Request for Comments (Proposed Standard) 2246, Internet Engineering Task Force, January 1999.
- [7] W. Diffie and M.E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, Nov 1976.
- [8] W. Diffie, P.C. van Oorschot, and M.J. Wiener. Authentication and Authenticated Key Exchanges. *Designs, Codes and Cryptography*, 2:107–125, 1992.

- [9] D. Eastlake, 3rd, and C. Kaufman. Domain name system security extensions. Request for Comments (Proposed Standard) 2065, Internet Engineering Task Force, January 1997.
- [10] N. Haller. The S/KEY one-time password system. Request for Comments (Informational) 1760, Internet Engineering Task Force, February 1995.
- [11] Neil M. Haller. the s/key one-time password system. In *Proceedings of the ISOC Symposium on Network and Distributed System Security*, 1994.
- [12] D. Harkins and D. Carrel. The internet key exchange (IKE). Request for Comments (Proposed Standard) 2409, Internet Engineering Task Force, November 1998.
- [13] P. Karn and W. Simpson. Photuris: Session-key management protocol. Request for Comments (Experimental) 2522, Internet Engineering Task Force, March 1999.
- [14] S. Kent and R. Atkinson. IP authentication header. Request for Comments (Proposed Standard) 2402, Internet Engineering Task Force, November 1998.
- [15] S. Kent and R. Atkinson. IP encapsulating security payload (ESP). Request for Comments (Proposed Standard) 2406, Internet Engineering Task Force, November 1998.
- [16] S. Kent and R. Atkinson. Security architecture for the internet protocol. Request for Comments (Proposed Standard) 2401, Internet Engineering Task Force, November 1998.
- [17] A. Keromytis and N. Provos. The use of HMAC-RIPEMD-160-96 within ESP and AH. Internet Draft, Internet Engineering Task Force, February 1999. Work in progress.
- [18] A. D. Keromytis, J. Ioannidis, and J. M. Smith. Implementing IPsec. In *Proceedings of Global Internet (GlobeCom) '97*, pages 1948 – 1952, November 1997.
- [19] RSA Laboratories. *PKCS #1: RSA Encryption Standard*, version 1.5 edition, 1993. November.
- [20] C. Madson and R. Glenn. The use of HMAC-MD5-96 within ESP and AH. Request for Comments (Proposed Standard) 2403, Internet Engineering Task Force, November 1998.
- [21] C. Madson and R. Glenn. The use of HMAC-SHA-1-96 within ESP and AH. Request for Comments (Proposed Standard) 2404, Internet Engineering Task Force, November 1998.
- [22] D. Maughan, M. Schertler, M. Schneider, and J. Turner. Internet security association and key management protocol (ISAKMP). Request for Comments (Proposed Standard) 2408, Internet Engineering Task Force, November 1998.
- [23] D. McDonald, C. Metz, and B. Phan. PF\_KEY Key Management API, Version 2. Request for Comments (Informational) 2367, Internet Engineering Task Force, July 1998.
- [24] S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer. Kerberos authentication and authorization system. Technical report, MIT, December 1987.
- [25] R. T. Morris. A Weakness in the 4.2BSD Unix TCP/IP Software. Computing Science Technical Report 117, AT&T Bell Laboratories, February 1985.
- [26] Data Encryption Standard, January 1977.
- [27] Digital Signature Standard, May 1994.
- [28] Secure Hash Standard, April 1995. Also known as: 59 Fed Reg 35317 (1994).
- [29] D. Piper. The internet IP security domain of interpretation for ISAKMP. Request for Comments (Proposed Standard) 2407, Internet Engineering Task Force, November 1998.
- [30] J. Postel. Domain name system structure and delegation. Request for Comments (Informational) 1591, Internet Engineering Task Force, March 1994.
- [31] Jon Postel. Internet Protocol. Internet RFC 791, 1981.
- [32] Jon Postel. Transmission Control Protocol. Internet RFC 793, 1981.
- [33] Niels Provos and David Mazières. A Future-Adaptable Password Scheme. In *Proceedings of the Annual USENIX Technical Conference*, 1999.
- [34] R. Rivest. The MD5 Message-Digest Algorithm. Internet RFC 1321, April 1992.
- [35] Bruce Schneier. Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish). In *Fast Software Encryption, Cambridge Security Workshop Proceedings*, pages 191–204. Springer-Verlag, December 1993.
- [36] William Simpson. IP in IP Tunneling. Internet RFC 1853, October 1995.
- [37] Secure Syslog. <http://www.core-sdi.com/Core-SDI/english/slogging/ssyslog.html>.
- [38] T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne, and S. Lehtinen. SSH protocol architecture. Internet Draft, Internet Engineering Task Force, February 1999. Work in progress.



## Minding Your Own Business

The Platform for Privacy  
Preferences Project  
and Privacy Minder

.....

Lorrie Faith Cranor  
AT&T Labs-Research

<http://www.research.att.com/~lorrie/>

June 1999

3

## Revealing Personal Info

### ■ Advantages

- home delivery of products
- customized information and services
- ability to buy things on credit

### ■ Disadvantages

- info might be used in unexpected ways
- info might be disclosed to other parties

2

## User Empowerment Approach

Develop tools that allow people  
to control the use and  
dissemination of their personal  
information

## Empowerment Tools

- Prevent your actions from being linked to you  
Crowds - AT&T Labs; The Anonymizer - anonymizer.com
- Allow you to develop persistent relationships  
not linked to each other or you  
Lucent Personal Web Assistant - Bell Labs
- Make informed choices about how your  
information will be used  
Platform for Privacy Preferences Project - W3C
- Know that assurances about information  
practices are trust worthy  
TRUSTe - Electronic Frontier Foundation and CommerceNet

4

## Platform for Privacy Preferences Project (P3P)

A framework for automated privacy discussions under development by W3C

- Services communicate about practices
- Users exercise preferences over those practices
- User agent can facilitate automated decision making, prompt user, exchange data, etc.

5

## Simplifying Notice and Choice

- **visual labels**
  - example: TRUSTe
- **machine readable labels**
  - example: Platform for Internet Content Selection (PICS)

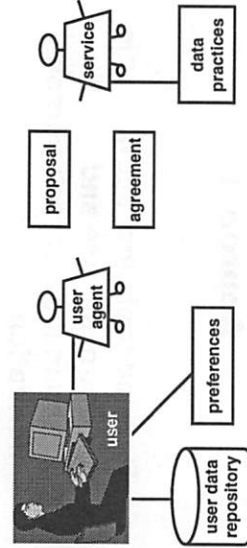
6

## Beyond Labeling

- Labels support *notice*, but provide only limited support of *choice*
- P3P supports *choice* by supporting
  - Multiple privacy policies
  - Explicit agreements (or rejection of proposed privacy policy)
  - Single-round "negotiation"

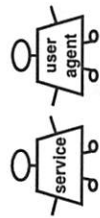
7

## Basic P3P Concepts



8

## A Simple P3P Conversation



User agent: Get index.html

Service: Here is my P3P proposal - I collect click-stream data and computer information for web site and system administration and customization of site

User agent: OK, I accept your proposal

Service: Here is index.html

9

## Data

- Referenced by category or element
  - Vocabulary includes 10 data categories
- Base data set includes elements all implementations should know about
- Services may create their own elements
- "P3P methods" may be used to transfer data referenced by element
  - Coupling between privacy disclosure and data collection

11

## Other Possible P3P Conversations

- Service offers choice of proposals
- Upon agreement, user agent automatically sends requested data
- No agreement is reached

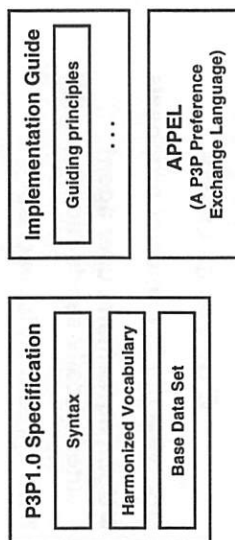
10

## Data Repository

- Users can store elements they don't mind providing to some services
- Services can gain access to stored elements through P3P agreements
- Elements can be automatically retrieved from repository when P3P methods or auto-fill forms are used

12

## W3C P3P Documents



13

## APPEL

- A rule language that expresses what should be done with P3P proposals
- Not essential to P3P, but useful for:
  - Sharing and installation of rule sets
  - Communicating to agents, search engines, proxies, or other servers
  - Portability between products
- Could be replaced by XML or RDF query language

15

## Guiding Principles

A statement of intent by members of the P3P working groups and a recommendation on how to use P3P to maximize privacy

- Information Privacy
- Notice and Communication
- Choice and Control
- Fairness and Integrity
- Security

14

## P3P Proposal

- A web site encodes its privacy practices in the form of a P3P proposal
- Automated tools can be used to do the actual encoding
- User agents are expected to translate information in proposals into a more user friendly format

16

## Types of Assertions

Proposals can contain 2 types of assertions:

- proposal level: assertions that apply generally to the whole proposal
  - "we are a member of TRUSTe"
- statement level: assertions that apply to a specific type of data
  - "we collect information about your computer for web site and system administration"

17

## P3P Implementation and Deployment

- Need user agent and server implementations
- Need Web sites to create P3P proposals
- Web sites can use P3P without a special server, but P3P-compliant server and tools allow them to take advantage of choice mechanisms

19

## Assertions that can be made in a P3P Proposal

- | <u>Proposal level</u> | <u>Statement level</u>         |
|-----------------------|--------------------------------|
| ■ Entity              | ■ Consequence                  |
| ■ Realm               | ■ Data category and/or element |
| ■ Disclosure URI      | ■ Purpose                      |
| ■ Access              | ■ Identifiable use             |
| ■ Assurance           | ■ Recipients                   |
| ■ Other disclosures   |                                |
| ● Change agreement    |                                |
| ● Retention           |                                |

18

## AT&T P3P Implementations

- P3P proposal generator
  - generates P3P proposal and human-readable policy from web-based questionnaire
  - written in Perl and implemented as a CGI script
- Privacy Minder
  - a P3P user agent
  - written in Java as a client-side proxy

20

## **Privacy Minder Demo**

21

## **Resources and Feedback**

- For further info on P3P see:  
<http://www.w3.org/P3P/>
- For AT&T P3P implementations and papers see:  
<http://www.research.att.com/projects/p3p/>
- Send your comments to  
[p3p-comments@w3.org](mailto:p3p-comments@w3.org) or discuss with a  
P3P working group member

22

# Trapeze/IP: TCP/IP at Near-Gigabit Speeds

*Andrew Gallatin, Jeff Chase, and Ken Yocum\**

Department of Computer Science

Duke University

{gallatin, chase, grant}@cs.duke.edu

## Abstract

This paper presents experiences with high-speed TCP/IP networking on a gigabit-per-second Myrinet network, using a Myrinet messaging system called Trapeze. We explore the effects of common optimizations above and below the TCP/IP protocol stack, including zero-copy sockets, large packets with scatter/gather I/O, checksum offloading, message pipelining, and interrupt suppression. Our experiments use extended FreeBSD 4.0 kernels on a range of Intel and Compaq Alpha platforms. These experiments give a snapshot of the FreeBSD TCP/IP implementation running at bandwidths as high as 956 Mb/s. We also report some results using Gigabit Ethernet products from Alteon Networks, which yielded a TCP bandwidth of 988 Mb/s using zero-copy sockets on a 500 MHz Compaq Alpha 21264 workstation.

## 1 Introduction

Over the next few years, new high-speed network standards — primarily Gigabit Ethernet — will consolidate an order-of-magnitude gain in network performance already achieved with specialized cluster interconnects such as Myrinet and SCI. As these technologies gain acceptance in LANs and server farms, they will place new performance pressure on network software. Although the latest desktop-class computers are capable of outstanding I/O performance, there is little quantitative basis to: (1) predict the performance they will actually deliver using standard TCP/IP networking on the new generation of networks, (2) quantify the importance of proposed optimizations (e.g., Jumbo Frames, zero-copy buffering, checksum offloading) to achieving the potential hardware performance, or (3) judge when alternatives such as user-level networking (e.g., VIA) are jus-

tified. In most cases published performance results are based on research prototypes using previous-generation technology.

This paper presents experiences with high-speed TCP/IP networking on a gigabit-per-second Myrinet network [3]. Our work is based on the Trapeze messaging system [10, 5, 1, 9], which consists of a messaging library and custom firmware for Myrinet. Using Trapeze firmware, Myrinet delivers communication performance at the limit of I/O bus speeds on many platforms, closely approaching the full gigabit-per-second wire speed on the most powerful hosts. This makes Trapeze/Myrinet a good vehicle for probing the limits of both the hardware and the networking software. In the experiments presented here, we exercised a Trapeze/Myrinet network with a network device driver supporting a standard kernel-based TCP/IP protocol stack on a range of DEC Alpha and Intel-based platforms. Our purpose is to provide a quantitative snapshot of the current state of the art for point-to-point TCP/IP communication on short-haul networks with low error rates, low latency, and gigabit-per-second bandwidth.

The kernel used in our experiments is FreeBSD 4.0, a descendent of the Berkeley 4.4 BSD code base, which incorporates several years worth of TCP/IP refinements. It is now widely accepted that current TCP implementations are capable of delivering a high percentage of available link speeds with large transfers, reflecting the success of these earlier efforts. However, on gigabit-per-second networks the performance of even the best TCP/IP implementations is dependent on key optimizations for low-overhead data movement, both above and below the protocol stack. One goal of this paper is to provide quantitative data to support insights into the effects and importance of these optimizations on current workstation/PC technology.

This paper outlines the key optimizations important for obtaining hardware potential from TCP/IP, their implementation in the network interface, network driver,

\*This work is supported by the National Science Foundation (CCR-96-24857, CDA-95-12356, and EIA-9870724) and equipment grants from Intel Corporation and Myricom.

and kernel socket code, and their effect on delivered TCP bandwidth, UDP latency, and CPU utilization at the sender and receiver. Below the TCP/IP stack, the Trapeze NIC firmware and network driver support page-aligned payload reception, interrupt suppression, large frames (MTUs), TCP checksum offloading, and an adaptive message pipelining scheme that balances low latency and high bandwidth. Above the protocol stack, at the socket layer, we have implemented new kernel support for zero-copy data movement in TCP as an extension to a zero-copy stream interface implemented by John Dyson. We show the effect of each of these factors on TCP/IP networking performance. We also report some results using similar features with Gigabit Ethernet adapters and switches from Alteon Networks.

Using Trapeze/Myrinet with zero-copy sockets, *netperf* attained a peak point-point bandwidth close to the link speed at 956 Mb/s on a 500 MHz Alpha 21264 PC platform equipped with prototype LANai-5 adapters from Myricom. At this speed, bandwidth is limited by the LANai-5 CPU. Newer controllers with upgraded CPUs promise still higher bandwidths. In fact, we measured bandwidth of 988 Mb/s on the same platform over the Alteon network, which uses a faster CPU on the adapters. The previous point-to-point record reported at *netperf.org* was 750 Mb/s, measured on a pair of mainframe-class SMP servers interconnected by HiPPI. We are not aware of any better result on public record.

This paper is organized as follows. Section 2 gives an overview of the Trapeze network interface, and Section 2.2 outlines the various optimizations for low-overhead TCP/IP communication. Section 3 presents performance results. We conclude in Section 4.

## 2 TCP/IP with Trapeze/Myrinet

This section presents background material important for understanding the performance results in Section 3. We first give an overview of the Trapeze messaging system, with a focus on the features relevant to TCP/IP networking. We then outline the optimizations used above and below the TCP/IP protocol stack to reduce data movement overhead and per-packet handling costs.

### 2.1 Trapeze Overview

The Trapeze messaging system consists of two components: a messaging library that is linked into the kernel or user programs, and a firmware program that runs on the Myrinet network interface card (NIC). The Trapeze firmware and the host interact by exchanging commands and data through a block of memory on the NIC, which is addressable in the host's physical address space using

programmed I/O. The firmware defines the interface between the host CPU and the network device; it interprets commands issued by the host and controls the movement of data between the host and the network link. The host accesses the network using macros and procedures in the Trapeze library, which defines the lowest level API for network communication across the Myrinet. Since Myrinet firmware is customer-loadable, any Myrinet site can use Trapeze.

Trapeze was designed primarily to support fast kernel-to-kernel messaging alongside conventional TCP/IP networking. Trapeze currently hosts kernel-to-kernel RPC communications and zero-copy page migration traffic for network memory and network storage, a user-level communications layer for MPI and distributed shared memory, a low-overhead kernel logging and profiling system, and TCP/IP device drivers for FreeBSD and Digital UNIX. These drivers allow a native TCP/IP protocol stack to use a Trapeze network through the standard BSD *ifnet* network driver interface. Figure 1 depicts this structure.

Trapeze messages are short *control messages* (maximum 128 bytes) with optional attached *payloads* typically containing application data not interpreted by the networking system, e.g., file blocks, virtual memory pages, or a TCP segments. The data structures in NIC memory include two message rings, one for sending and one for receiving. Each message ring is a circular array of 128-byte control message buffers and related state, managed as a producer/consumer queue shared with the host. From the perspective of a host CPU, the NIC produces incoming messages in the receive ring and consumes outgoing messages in the send ring.

Trapeze has several features useful for high-speed TCP/IP networking:

- **Separation of header and payload.** A Trapeze control message and its payload (if any) are sent as a single packet on the network, but the control message and payload are handled separately by the firmware and message system. In particular, payloads are transferred to and from aligned page frames of host memory, which the driver allocates from the virtual memory page pool. This enables the zero-copy optimizations described in Section 2.2.1, assuming the driver is able to place the TCP/IP headers in the control message portion of the packet.
- **Large MTUs with scatter/gather DMA.** Since Myrinet has no fixed maximum packet size (MTU), the maximum payload size of a Trapeze network is easily configurable. Trapeze supports scatter/gather DMA so that payload buffers may span multiple noncontiguous page frames. Scatter/gather allows

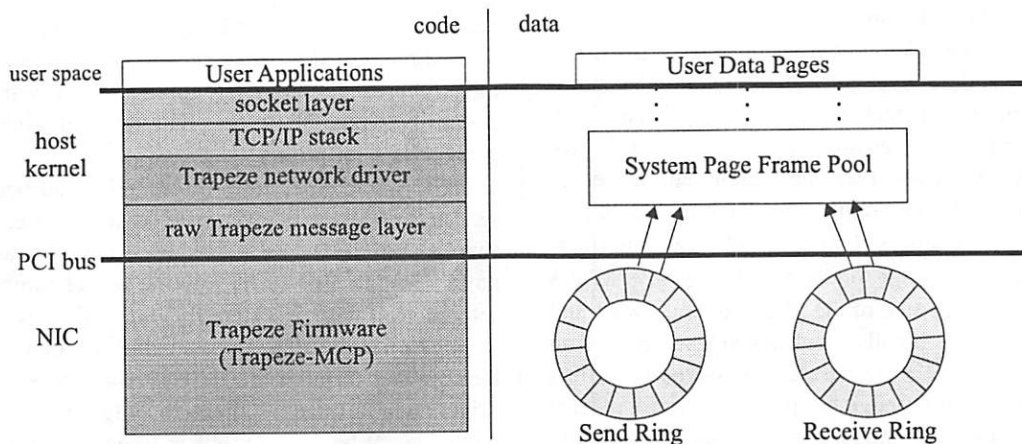


Figure 1: Using a Trapeze endpoint for kernel-based TCP/IP networking.

us to run TCP with MTUs of 64 KB or larger, yielding very low per-packet overheads in the networking code.

- **Adaptive message pipelining.** The Trapeze firmware pipelines DMA transfers on the I/O bus and network link to minimize large packet latency [10]. The pipelining scheme adaptively reverts to larger unpipelined DMA transfers in bandwidth-constrained scenarios [9]. This technique enables Trapeze to combine low large-packet latencies with high bandwidth under load.

One item missing from this list is *interrupt suppression*. Handling of incoming messages is interrupt-driven when Trapeze is used from within the kernel; incoming messages are routed to the destination kernel module (e.g., the TCP/IP network driver) by a common interrupt handler in the Trapeze message library. Interrupt handling imposes a per-packet cost that becomes significant with smaller MTUs. Some high-speed network interfaces reduce interrupt overhead by amortizing interrupts over multiple packets during periods of high bandwidth demand. For example, the Alteon Gigabit Ethernet NIC includes support for adaptive interrupt suppression, selectively delaying packet-receive interrupts if more packets are pending delivery. Trapeze implements interrupt suppression for a lightweight kernel-kernel RPC protocol [1], but we do not use receive-side interrupt suppression for TCP/IP because it yields little benefit for MTUs larger than 16KB at current link speeds.

## 2.2 Low-Overhead Data Movement

This section describes the optimizations used above and below the TCP/IP protocol stack to reduce data movement overhead for copying and checksumming data. These overheads increase with the volume of data moved per unit of time; at gigabit-per-second bandwidths, data movement overhead can consume a large share of CPU cycles. Unfortunately, faster CPUs do not help appreciably because copying is memory-intensive.

We describe the data movement optimizations as extensions to the conventional FreeBSD send/receive path, which is based on variable-sized kernel network buffers called *mbufs* [8]. Standard *mbufs* contain their own buffer space, while *external mbufs* hold pointers to other kernel buffers, e.g., file buffers or the virtual memory page frames used as Trapeze payload buffers. Packet data is stored in linked chains of *mbufs* passed between levels of the system; the TCP/IP protocol stack adds and removes headers and checksums by manipulating the *mbufs* in the chain. On a normal transmission, the socket layer copies IP message from a user memory buffer into a chain, which is passed through the TCP/IP stack to the network driver. On the receiving side, the driver constructs a chain containing each incoming packet header and payload, and passes the chain through the TCP/IP stack to the socket layer. When the receiving process accepts the data, e.g., with a *read* system call, a socket-layer routine (*soreceive*) copies the payload into user memory and frees the kernel *mbuf* chain.

### 2.2.1 Zero-Copy Sockets

Conventional TCP/IP communication incurs a high cost to copy data between kernel buffers and user process virtual memory at the socket layer. This situation has motivated development of techniques to reduce or eliminate data copying by *page remapping* between the user process and the kernel when size and alignment properties allow [6, 4, 7]. A page remapping scheme should preserve the copy semantics of the existing socket interface.

In general, zero-copy optimizations assume MTUs matched to the page size of the endstation hardware and operating system. Ideally, each packet payload is an even multiple of the page size, and is stored in buffers that naturally align on page boundaries. On the receive side, the NIC must separate the headers and payload into separate buffers, leaving the payload page-aligned. This can be done with special support to recognize TCP/IP packets on the NIC, or by constructing receive *mbuf* chains that optimistically assume that received packets are TCP packets. In Trapeze, the sending host explicitly separates header and payload portions of each packet; the Trapeze driver optimistically assumes that data in the first *mbuf* of an outgoing chain is header data, and places its data in the control message. The link layer preserves this separation on the receiving side.

We implemented zero-copy TCP/IP extensions at the socket layer in the FreeBSD 4.0 kernel, using code developed by John Dyson for zero-copy I/O through the *read/write* system call interface. The zero-copy extensions require some buffering support in the network driver, but are otherwise independent of the underlying network, assuming that it supports sufficiently large MTUs and page-aligned sends and receives. Section 3 reports results from zero-copy TCP experiments on both Trapeze/Myrinet and Alteon Gigabit Ethernet hardware.

The page remapping occurs in a variant of the *uiomove* kernel routine, which directs the movement of data to and from the process virtual memory for all variants of the I/O read and write system calls. Our zero-copy socket code is implemented as a new case alongside Dyson's code in *uiomoveco*, which is invoked from socket-layer *sosend* and *soreceive* when a process requests the kernel to transfer a page or more of data to or from a page-aligned user buffer.

For a zero-copy read, *uiomoveco* maps kernel buffer pages directly into the process address space. If the *read* is from a file, it creates a copy-on-write mapping to a page in FreeBSD's unified buffer cache; the copy-on-write preserves the file data in case the user process stores to the remapped page. In the case of a receiver read from a socket, copy-on-write is unnecessary because there is no need to retain the kernel buffer after the read; ordinarily *soreceive* simply frees the kernel

buffers once the data has been delivered to the user process. The remapping case instead frees just the *mbuf* headers and any physical page frames that previously backed remapped virtual pages in the user buffer. Thus most receive-side page remappings actually trade page frames between the process and the kernel buffer pool, preserving equilibrium.

On the send side, copy-on-write is used because the sending process may overwrite its send buffer once the send is complete. The send-side code maps each whole page from the user buffer into the kernel address space, references it with an external *mbuf*, and marks the page as copy-on-write. The *mbuf* chains and their pages are then passed through the TCP/IP stack to the network driver, which attaches them to outgoing messages as payloads. When each *mbuf* is freed on transmit complete, the external free routine releases the page's copy-on-write mapping. The new socket layer code handles only anonymous virtual memory pages; we do not support zero-copy transmission of memory backed by mapped files because this would duplicate the functionality of the *sendfile* routine already implemented by David Greenman.

### 2.2.2 Checksum Offloading

Checksum offloading eliminates host-side checksumming overheads by performing checksum computation with hardware assist in the NIC. TCP/IP checksum offloading is supported by Myricom's recently released LANai-5 adapter, and by other high-speed network interfaces including Alteon's Gigabit Ethernet NICs based on the Tigon-II chipset.

The NIC and the host-side driver must act in concert to implement checksum offloading. The LANai-5 and Alteon NICs support checksum offloading in the host-PCI DMA engine, which computes the raw 16-bit ones-complement checksum of each DMA transfer as it moves data to and from host memory. Using this checksum need not demand any significant change to the IP stack: simply setting a *M\_HWCKSUM* flag in the header of an *mbuf* chain bypasses the software checksum computation in *in\_cksum*. However, using hardware checksumming for IP protocol family is complicated by three factors:

- Movement of each packet may occur in multiple DMA transfers to or from distinct host memory buffers. If the hardware makes each partial checksum available to the NIC firmware separately (as Myricom's LANai-5 NIC), then firmware and/or host software must combine these partial checksums (using one's complement addition) to obtain a complete checksum.

- TCP or UDP checksumming actually involves two checksums: one for the IP header (including fields overlapping with the TCP or UDP header) and a second end-to-end checksum covering the TCP or UDP header and packet data. In a conventional system, TCP or UDP computes its end-to-end checksum before IP fills in its overlapping IP header fields (e.g., options) on the sender, and after the IP layer restores these fields on the receiver. Checksum offloading involves computing these checksums below the IP stack; thus the driver or NIC firmware must partially dismantle the IP header in order to compute a correct checksum.
- Since the checksums are stored in the headers at the front of each IP packet, a sender must complete the checksum before it can transmit the packet headers on the link. If the checksums are computed by the host-NIC DMA engine, then the last byte of the packet must arrive on the NIC before the firmware can determine the complete checksum.

Trapeze currently supports TCP checksum offloading only on LANai-5 receivers. Checksum offloading is not supported on the sending side, in part because Trapeze uses message pipelining to minimize latency of large packets. With message pipelining the front of a packet may be transmitted on the link before the tail of the packet arrives on the NIC, and therefore before the checksum can be determined. One solution is to append the end-to-end checksum to the tail of the outgoing packet; while this would depart from the standard IP packet format, it is transparent to the end hosts because the Trapeze firmware and driver can reconstruct the packet at the receiving side. Of course, this approach would compromise interoperability in a standards-based network containing some endstations that do not support checksum offloading. The alternative, apparently implemented in Alteon's NICs, is to use store-and-forward packet transmission at the sender, which increases large-packet latencies (see Section 3.4).

Trapeze uses the NIC DMA engines to checksum packet data, but header checksums are computed by special-case code for TCP/IP in the Trapeze network driver. The Trapeze firmware combines partial checksums for all DMA operations on the payload portion of the message, then passes the partial checksum to the host-side driver through a logical control register. The driver then computes an IP header checksum, computes the layer-4 header checksum using a scratch copy of the IP header, combines the layer-4 header checksum with the payload checksum to determine the complete end-to-end checksum, and compares the computed checksums with those transmitted in the packet. Our philosophy is that any instructions that manipulate the IP

header should be executed on the fast host CPU rather than on the NIC. In contrast, the Alteon NICs perform both header and data checksums in the NIC firmware.

### 3 Experimental Results

We ran our experiments on four Intel and Alpha hardware configurations:

- **Pentium-II/440LX.** These are Dell Dimension XPS D-300 workstations containing a 300 MHz Intel Pentium-II processor and an Intel 440LX chipset. Each machine has 128MB of RAM and a Myricom Lanai 4.1 SAN adapter (M2M-PCI32C) connected to a 32-bit 33 MHz PCI slot.
- **Pentium-II/440BX.** These machines use a Pentium-II processor clocked at 450 MHz on an Asus P2B motherboard with an Intel 440BX chipset. Each machine has 128MB of RAM and a Myricom Lanai 4.1 LAN adapter (M2F-PCI32) connected to a 32-bit 33 MHz PCI slot.
- **DEC Miata.** These are Digital Personal Workstation 500au platforms with a 500MHz Alpha 21164 CPU, a 96KB L2 cache, a 2MB L3 cache, and the Digital 21174 "Pyxis" chipset. These machines are configured with 512MB of RAM and a Myricom Lanai 4.1 SAN adapter connected to a 32-bit 33 MHz PCI slot. The Pyxis limits I/O bandwidth to approximately 103 MB/s on the sending side.
- **DEC Monet.** These are Compaq XP1000 Professional Workstations, with a 500 MHz Alpha 21264 CPU, a 4MB L2 cache, and the Digital 21272 "Tsunami" chipset. These machines are configured with 640MB of RAM and a Myricom Lanai 5.2 SAN adapter connected to a 64-bit 33 MHz PCI slot. The Lanai-5 is described in <http://www.myri.com:80/scs/PCI64X/PCI64X-spec.html>. We also report some Gigabit Ethernet measurements from this platform, using Alteon ACENIC adapters based on the Tigon-II chipset (firmware revision 12.3.8), interconnected through an ACEswitch 1080 (firmware revision 5.0.24).

All systems run kernels built from the same FreeBSD 4.0 source pool, which was current as of 4/15/99. The hosts are interconnected through diverse Myrinet switch models, which have no measurable effects on the results.

To take timings, we used *netperf* version 2.1pl3 built from the FreeBSD ports collection. We modified *netperf* to collect CPU utilization by reading the system timers directly from kernel memory via *libkvm*, in order to correctly charge interrupt overhead to the *netperf* process.

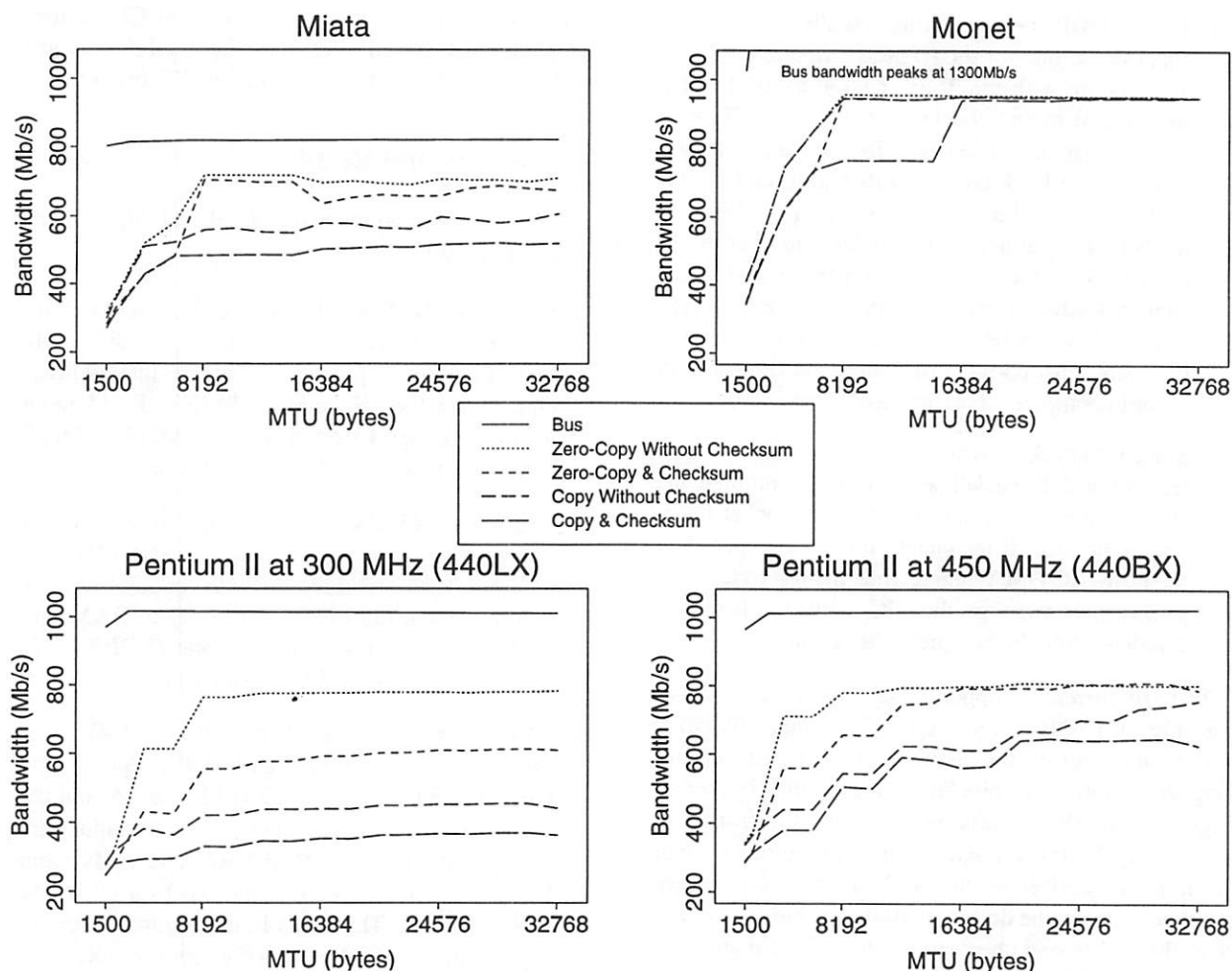


Figure 2: TCP Bandwidth

All tests were run on isolated machines, and the vast majority of the interrupts serviced came from the gigabit NIC.

### 3.1 TCP Bandwidth

We measured unidirectional point-to-point TCP bandwidth using *netperf-l60-C-c*, which sends as much data as it can in 60 seconds, then computes the average bandwidth over the interval. Socket buffers were set to 512 KB for all tests with an MTU of 4KB or greater. Tests with smaller MTUs used 64 KB socket buffers. Note that the *netperf* sender and receiver do not access the data in these tests.

Figure 2 shows the average Trapeze TCP bandwidth on all four platforms as a function of the MTU. To show the effects of copying and checksumming, we tested with the zero-copy optimizations enabled and disabled, and with checksumming enabled and disabled. All checksumming is done in software except on the receiver

side of the Monet configuration, which uses checksum offloading on the LANai-5 adapter as described in Section 2.2.2.

The graphs show the bandwidth costs of copying and checksumming, primarily on the older platforms with less memory system bandwidth. The effect is most apparent on the P-II/440LX, which is capable of a peak bandwidth of only 450 Mb/s if it is forced to copy the data, while peak bandwidth almost doubles to 780 Mb/s when the zero-copy optimizations are enabled. The cost of checksumming is more pronounced with zero-copy enabled, since the checksum code must bring the data into the CPU cache. On the P-II/440BX, superior memory system bandwidth allows the system to achieve close to its peak bandwidth even while copying or checksumming, but not both, and only for very large MTUs when the CPU is not already busy with packet-handling overheads. This is also visible on the Miata, which has comparable memory system bandwidth, but the effect is less pronounced because the I/O bus limits the achievable

bandwidth. The Monet has adequate memory system bandwidth to deliver a peak bandwidth of 956 Mb/s for sufficiently large MTUs, even while copying and checksumming. Even so, copying and checksumming have a significant effect on the available CPU cycles remaining to process the data at these speeds, as Section 3.2 shows.

Figure 2 also shows the difficulty of achieving high bandwidth using the small, 1500-byte MTUs of the Gigabit Ethernet standard. In addition to increasing packet handling overheads, small MTUs defeat the zero-copy optimizations. The combined effect causes the host CPU to saturate at bandwidths as low as 300 Mb/s, and none of the platforms is capable of using more than half of the available link speed. Section 3.2 examines the overheads in more detail. All platforms are capable of achieving most of their peak bandwidth at MTUs large enough to contain a TCP/IP header and a page of data; the Intel platform bandwidths rise faster because zero-copy kicks in at the 4KB page size, while the Alpha platforms use an 8KB page size.

While we were pleased with the Trapeze bandwidth results on Monet, which we believed to be an open-source record, we measured even higher bandwidths with Alteon's new Gigabit Ethernet products. The Monet delivers point-to-point TCP bandwidth of 988 Mb/s with zero-copy sockets over Alteon. The higher bandwidth is apparently due to lower overheads in the Alteon controller, which sports dual 100 MHz MIPS R4000-like processors delivering several times the processing capacity of the LANai-5 CPU. We anxiously await the LANai-7 from Myricom.

### 3.2 CPU Utilization

The potential for high bandwidth has little value in practice if communication overheads leave no CPU power to process the data. CPU utilization is just as important as bandwidth, since bandwidths will drop if application processing saturates the CPU. All the optimizations we explore are fundamentally directed at reducing overhead; they increase the delivered bandwidth only indirectly by delaying saturation of the host CPUs. Overheads are reduced by reducing packet handling costs with larger MTUs, reducing interrupt costs with larger MTUs or interrupt suppression, and reducing data-touching costs through zero-copy page remapping or checksum offloading.

Figures 3 and 4 show the average CPU utilizations on the sender and receiver respectively for the bandwidth tests reported in Section 3.1. Some of the results vary noticeably due to several factors. On the receiver, this may be affected by incomplete support for page coloring in the Alpha FreeBSD port; the runs show a bimodal distribution on the Alpha receiver configurations when

copying is used. In the zero-copy sender results, irregularities result when *netperf* reuses a send buffer page before the driver determines that the previous transmit on that buffer is complete. The Trapeze driver detects this case and conservatively copies the page, although *netperf* does not actually store to the buffer in this experiment; if the process did store to the page then a copy-on-write would result. The sender-side zero-copy optimizations trigger with varying probabilities on different configurations, and are affected by CPU speed, the process send buffer size, and the Trapeze ring size, given that Trapeze suppresses transmit-complete notices until a send ring entry is reused. Some step behavior results from the TCP implementation selecting packet sizes that are integral multiples of the page size for odd MTUs; these effects are less pronounced on the Intel platforms, which use 4KB rather than 8KB pages. The numbers presented here are averages of 20 runs.

On the receiving side, all graphs show a trend of declining CPU utilization with large MTUs, with much lower CPU utilizations when data-movement costs such as copying and checksumming are eliminated. The downward trend with larger MTUs is most pronounced on the faster platforms, since the bandwidth of older platforms such as 440LX is initially limited by the CPU; reduced overheads result in higher bandwidth rather than lower CPU utilization. Similarly, CPU utilizations initially increase with larger MTUs on the sending side. This is because the larger MTUs allow higher bandwidth at the receiver, driving the sender to transmit faster. Once peak bandwidth is attained, the CPU utilizations begin to drop with increasing MTU. The graphs reflect the higher CPU costs on the receiving side, primarily due to lower interrupt overheads at the sender.

These graphs show that copying and checksumming optimizations are extremely important even on the platforms that are capable of achieving peak bandwidth without them. Any reduction in overhead translates directly into lower CPU utilization, leaving more cycles available for application processing at a given bandwidth. Note also that disabling checksumming yields little benefit on the Monet receiver because of checksum offloading: the small incremental CPU cost is due to checksumming the headers in the driver.

The receiver utilizations in Figure 3 again reinforce the importance of the Jumbo Frames standard promulgated by Alteon and Microsoft, which would increase the Gigabit Ethernet MTU to 9000 bytes. The Intel receiver CPUs are saturated at 1500-byte MTUs, showing that the bandwidth limitation near 300 Mb/s for standard Ethernet MTUs is due to receiver CPU saturation caused by the overhead of handling the larger number of packets. Slightly higher 1500-byte bandwidths are achieved on the Alphas due to the faster host CPUs: on

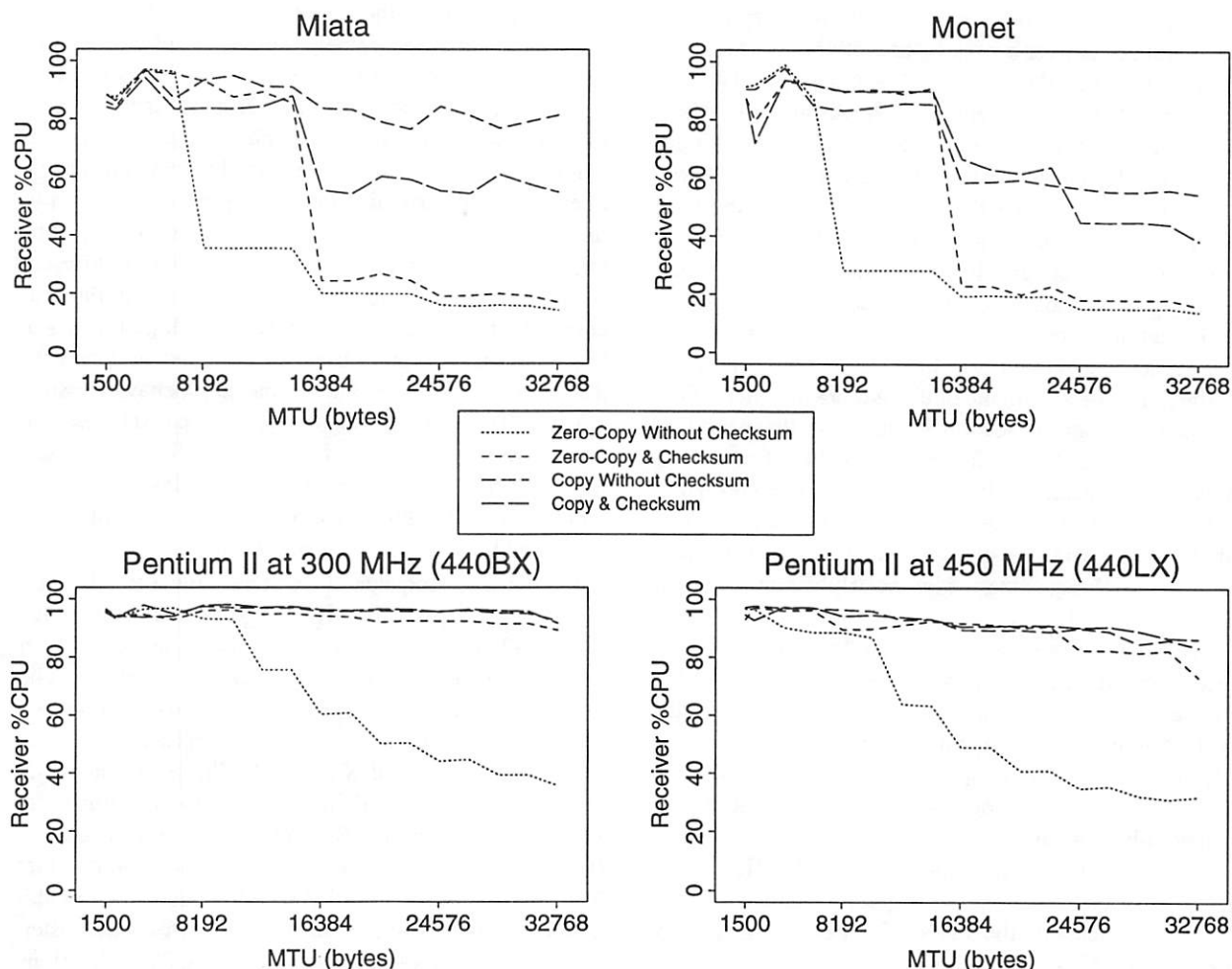


Figure 3: Receiver CPU Utilization

Miata the 1500-byte bandwidth is limited at 313 Mb/s by saturation of the CPU on the LANai-4 NIC, while the faster LANai-5 NIC delivers bandwidths closer to 410 Mb/s before saturating. However, at this speed packet handling overheads push the Monet's Alpha 21264 host CPU to 90% utilization, even while it is driving less than half of the link speed. The Monet's receiver utilization drops below 30% with 8KB packet sizes when zero-copy and checksum offloading are enabled, even as the delivered bandwidth more than doubles to 956 Mb/s.

Looking further, the results show that the 9000-byte MTU of the Jumbo Frames standard is sufficient to achieve near-peak bandwidth on all platforms. However, Figure 3 shows that if other host overheads are present, per-packet overheads can constrain peak bandwidth even if Jumbo Frames are used. The 440BX platform does not attain its peak bandwidth until 16KB MTUs, and does not achieve its minimal CPU utilization until the MTUs reach 57KB. Receiver CPU utilization on this platform drops from 88% to 48% as the MTU grows from 8KB to

16KB.

### 3.3 TCP Overhead

To better understand the costs responsible for the CPU utilizations presented in Figures 4 and 3, we used *iprobe* to derive a breakdown of receiver overheads on Miata for selected MTU sizes at bandwidth levels held in the 300-400 Mb/s range by a slow sender. *Iprobe* (Instruction Probe) is an on-line profiling tool developed by the performance group (High Performance Servers/Benchmark Performance Engineering) of Digital/Compaq. It uses the Digital Alpha on-chip performance counters to report detailed execution breakdowns with low overhead (3%-5%), using techniques similar to those reported in [2]. We gathered our data using a local port of *iprobe\_suite-4.0* to FreeBSD. This port will be integrated into the next release of *iprobe*.

Figure 5 shows the breakdown of receiver overhead into five categories: data movement overheads for copy-

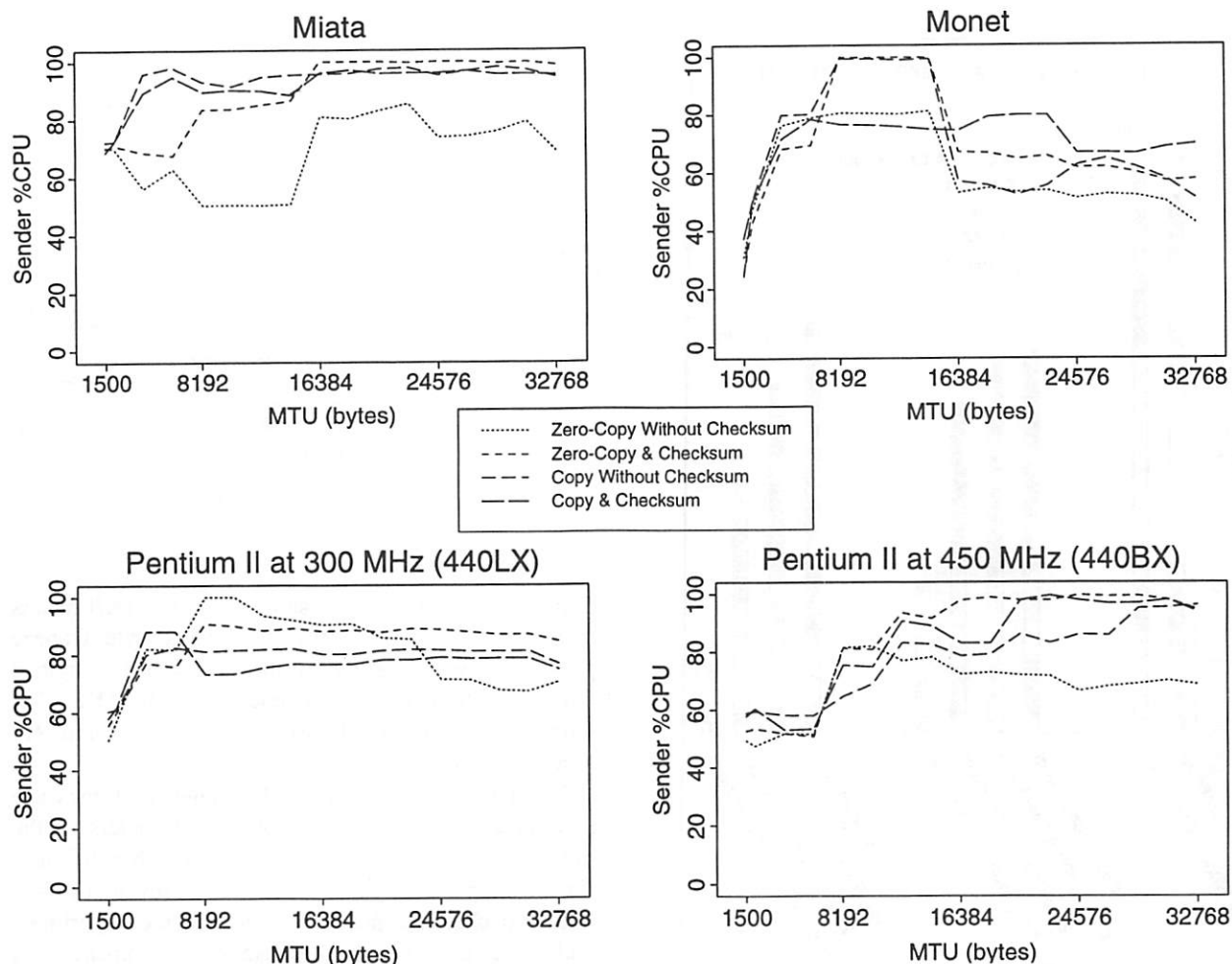


Figure 4: Sender CPU Utilization

ing and checksumming, interrupt handling, virtual memory costs (buffer page allocation and/or page remapping), Trapeze driver overheads, and TCP/IP protocol stack overheads. With a 1500-byte MTU the Miata is near 80% saturation at a bandwidth of 300 Mb/s. While the overhead can be reduced somewhat by checksum offloading and interrupt suppression, about 55% of CPU time is spent on unavoidable packet-handling overheads in the driver and TCP/IP stack, and data movement costs at the socket layer. With an 8KB payload, the bandwidth level has increased to 360 Mb/s, while CPU time spent in packet handling has dropped from 55% to 24%. Data movement overheads grow slightly due to the higher bandwidth, but the larger MTU introduces the opportunity to almost fully eliminate these overheads by enabling zero-copy optimizations. While the zero-copy optimization has some cost in VM page remapping, the reduced memory system contention causes other overheads to drop slightly, leaving utilizations in the 24% range if checksums are disabled (checksum offloading is

not supported on the LANai-4 NICs used in this experiment). Again, these measurements reinforce the inadequacy of the 1500-byte standard Ethernet for high-speed networking, and the importance of the Jumbo Frames standard.

The rightmost set of bars in Figure 5 shows the overhead breakdown for 57K MTUs at a bandwidth of 390 Mb/s. While data movement overheads increase slightly due to the higher bandwidth, these costs can be eliminated with page remapping, which increases VM overheads but again causes other non-VM overheads to drop slightly due to reduced memory system contention. In the zero-copy experiment, the larger MTU does not affect VM page remappings at all relative to the 8KB MTU, since these costs are proportional to the number of pages of data transferred. However, per-packet TCP/IP and driver overheads drop from 8% to just 3% of CPU, even as bandwidth increases by about 10%. The Miata can handle the 390 Mb/s of bandwidth with a comfortable 10% CPU utilization.

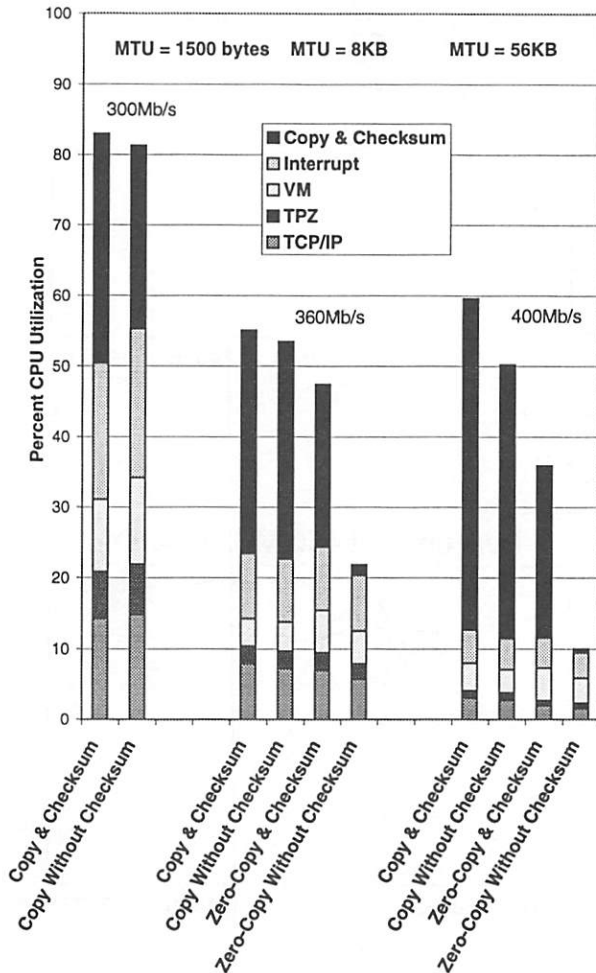


Figure 5: TCP Receiver CPU Utilization Breakdown

### 3.4 UDP Latency

Figure 6 shows the one-way UDP latency for various packet sizes using Trapeze and the Alteon Gigabit Ethernet on Monet. These results were obtained with *netperf -tUDP\_RR -l60*, which ping-pongs a packet of the requested size for one minute. We plotted points for one half of the average round-trip latency for one-byte packets and for packet sizes of 1KB to 8KB in 1KB increments. For these experiments we disabled checksum offloading on the Alteon after observing unexpectedly high latencies with checksum offloading enabled. The comparison is fair because Trapeze does not support checksum offloading on this platform. The two sets of lines for each configuration show the latency with software checksums and with checksums disabled. Zero-copy sockets are responsible for the lower latency in all configurations when the 8KB page size is reached.

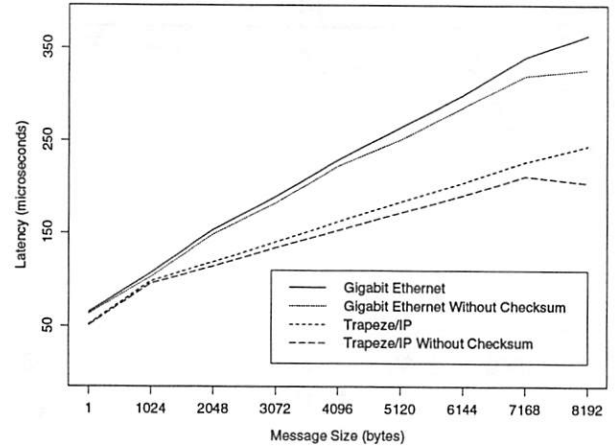


Figure 6: UDP One-Way Message Latency

In the Trapeze runs, the slope change at 1KB results from use of a payload buffer. The one-byte Trapeze packets are sent in a control message with no payload buffer, resulting in lower latency. For the 1KB-7KB sizes, the data is copied into a payload buffer and sent as a Trapeze payload.

The latency results show the benefits of message pipelining in Trapeze, which overlaps transfers on the link with transfers on the sender and receiver I/O bus. This overlap causes packet latencies to grow at a slower rate as packet size increases. In fact, the experiment understates these benefits because message pipelining is supported only on the receiver on the LANai-5 NIC used in this configuration, due to a change in the meaning of certain control registers on the LANai-5.

## 4 Conclusion

The experiments reported in this paper give a quantitative snapshot of the state of the art for TCP/IP networking performance on current-generation desktop-class PCs and workstations and gigabit-per-second networks.

Our measurements are taken from the standard high-quality TCP/IP implementation in the FreeBSD 4.0 kernel, supplemented with support for a range of techniques to reduce communication overheads. These include zero-copy sockets and several features implemented in the Trapeze firmware for Myrinet, including large MTUs with scatter/gather I/O, page-aligned payload buffers, adaptive message pipelining, interrupt suppression, and checksum offloading. With these features, we have measured TCP/IP bandwidths of 956 Mb/s using Trapeze/Myrinet and 988 Mb/s using an Alteon Gigabit Ethernet network. These are the highest TCP band-

widths on public record at present. The 500 MHz Alpha 21264 platform is capable of handling these bandwidths with CPU overheads as low as 20%.

## 5 Acknowledgments

The equipment used in this work was provided by the National Science Foundation, Intel Corporation, and Myricom Inc. We would like to thank Bob Felderman of Myricom for his frequent technical assistance, and Myricom for making LANai-5 prototypes available. We also thank Ted Schroeder and Alteon Networks for their support and for releasing Tigon-II documentation and firmware to research groups.

The FreeBSD port to the Alpha platforms is due to Doug Rabson. Don Rice at Compaq Workstation Engineering has given us invaluable assistance with the FreeBSD/Alpha Monet platform support. Bill Paul wrote the FreeBSD driver for the Tigon-II Gigabit Ethernet cards. We thank Phillip Ezolt and Digital/Compaq's High Performance Servers/Benchmark Performance Engineering group for making *iprobe* available as open source.

Jenise Swall introduced us to S-Plus and provided much-needed assistance in coercing this tool to produce our graphs.

## 6 Availability

Trapeze is available in source form with a BSD-style copyright from <http://www.cs.duke.edu/ari/trapeze>. Our *iprobe* port to FreeBSD/Alpha is available from <http://www.cs.duke.edu/ari/iprobe.html>. FreeBSD extensions (including Monet platform support, Alteon driver extensions, and zero-copy sockets) are incorporated into the FreeBSD code base or are available from the Trapeze Web site.

## References

- [1] Darrell Anderson, Jeffrey S. Chase, Syam Gadde, Andrew J. Gallatin, Kenneth G. Yocum, and Michael J. Feeley. Cheating the I/O bottleneck: Network storage with Trapeze/Myrinet. In *Proceedings of the 1998 Usenix Technical Conference*, June 1998. Duke University CS Technical Report CS-1997-21.
- [2] Jennifer Anderson, Lance Berc, Jeff Dean, Sanjay Ghemawat, Monika Henzinger, Shun-Tak Leung, Mark Vandevoorde, Carl Waldspurger, and Bill Weihl. Continuous profiling: Where have all the cycles gone? In *Proceedings of the Sixteenth*

*ACM Symposium on Operating System Principles (SOSP)*, October 1997.

- [3] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W-K Su. Myrinet - a gigabit-per-second local area network. *IEEE Micro*, February 1995.
- [4] José Carlos Brustoloni and Peter Steenkiste. Effects of buffering semantics on I/O performance. In *Proc. of the Second Symposium on Operating Systems Design and Implementation*, pages 277–291, Seattle, WA, October 1996. USENIX Assoc.
- [5] Jeffrey S. Chase, Andrew J. Gallatin, Alvin R. Lebeck, and Kenneth G. Yocum. Trapeze messaging API. Technical Report CS-1997-19, Duke University, Department of Computer Science, November 1997.
- [6] Zubin D. Dittia, Guru M. Parulkar, and Jerome R. Cox. The APIC approach to high performance network interface design: Protected DMA and other techniques. In *Proceedings of IEEE Infocom*, 1997. WUCS-96-12 technical report.
- [7] Hsiao-Keng and Jerry Chu. Zero-copy TCP in Solaris. In *Proceedings of the USENIX 1996 Annual Technical Conference*, January 1996.
- [8] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4BSD Unix Operating System*. Addison Wesley, Reading, MA, 1996.
- [9] Kenneth G. Yocum, Darrell C. Anderson, Jeffrey S. Chase, Syam Gadde, Andrew J. Gallatin, and Alvin R. Lebeck. Adaptive message pipelining for network memory and network storage. Technical Report CS-1998-10, Duke University Department of Computer Science, April 1998.
- [10] Kenneth G. Yocum, Jeffrey S. Chase, Andrew J. Gallatin, and Alvin R. Lebeck. Cut-through delivery in Trapeze: An exercise in low-latency messaging. In *Proceedings of the Sixth IEEE International Symposium on High Performance Distributed Computing (HPDC-6)*, pages 243–252, August 1997.



# Managing Traffic with ALTQ

Kenjiro Cho

*Sony Computer Science Laboratories, Inc.*

*Tokyo, Japan 1410022*

`kjc@csl.sony.co.jp`

## Abstract

ALTQ is a package for traffic management. ALTQ includes a queueing framework and several advanced queueing disciplines such as CBQ, RED, WFQ and RIO. ALTQ also supports RSVP and diffserv. ALTQ can be configured in a variety of ways for both research and operation. However, it requires understanding of the technologies to set up things correctly. In this paper, I summarize the design trade-offs, the available technologies and their limitations, and how they can be applied to typical network settings.

## 1 Queueing Basics

Essentially, every traffic management scheme involves queue management. A large number of queueing disciplines have been proposed to date in order to meet contradictory requirements such as fairness, protection, performance bounds, ease of implementation or administration.

### 1.1 Queueing Components

Figure 1 illustrates queueing related functional blocks on a router. Each functional block could be needed to build a certain service but is not always required for other services. In fact, most routers currently in use do not have all the functional blocks.

Packets arrive at one interface of the router (ingress interface), and then, are forwarded to another interface (egress interface). A router could have functional blocks in the ingress interface to police incoming packets but the main functional blocks reside in the egress interface. The function of each block is described below.

**Classifier** Packet classifiers categorizes packets based on the content of some portion of the packet header. (e.g., addresses and port numbers). Packets matching some specified rule are classified for further processing.

**Meters** Traffic meters measure the properties of a traffic stream (e.g., bandwidth, packet counts). The measured characteristics are stored as flow state and used by other functions.

**Markers** Packet markers set a particular value to some portion of the packet header. The written values could be a priority, congestion information, an application type, or other types of information.

**Droppers** Droppers discard some or all of the packets in a traffic stream in order to limit the queue length, or as an implicit congestion notification.

**Queues** Queues are finite buffers to store backlogged packets. A queueing discipline could have multiple queues for different traffic classes.

**Schedulers** Schedulers select a packet to transmit from the backlogged packets in the queue.

**Shapers** Shapers delay some or all of the packets in a traffic stream in order to limit the peak rate of the stream. A shaper usually has a finite-size buffer, and packets may be discarded if there is not sufficient buffer space to hold the delayed packets.

A queueing discipline is, in general, defined as a set of the functional blocks at the egress interface, and usually consists of a specific queue structure, a scheduling mechanism and a dropper mechanism. However, the functional blocks described here are conceptual and a wide variety of combinations are possible.

### 1.2 Queueing Disciplines

Bandwidth allocation is one of the most important goals of a queueing discipline. Fair or preferential bandwidth allocation can be achieved by using an appropriate queueing discipline. The same mechanism also isolates a misbehaving flow, and thus, protects other traffic.

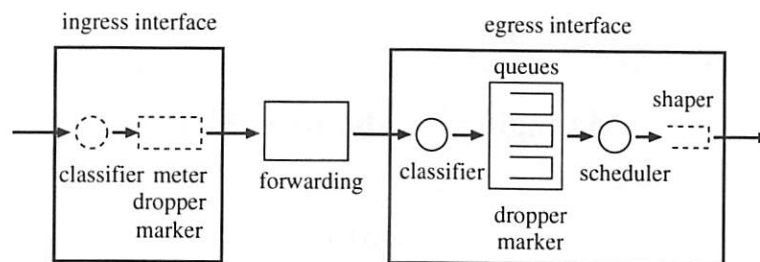


Figure 1: Queueing Architecture

Another important goal is to control delay and jitter that are critical to emerging real-time applications. It is possible to bound the delay and jitter of a flow by reserving the necessary network resources. Admission control is required to decide whether requested resources can be allocated. It is also needed to regulate the rate of the reserved flow by means of shaping. The incoming rate should be less than the reserved rate to avoid delay caused by the flow's own traffic. A leaky bucket is a simple shaper mechanism with a finite buffer size. Another popular shaper mechanism is a token bucket that allows small bursts with a configurable burst size. The token bucket can accommodate traffic streams with bursty characteristics so it is more suitable for the current Internet traffic.

Yet another goal of a queueing discipline is congestion avoidance. TCP considers packet loss as a sign of congestion. A router can notify TCP of congestion implicitly by intentionally dropping a packet.

The following list describes representative queueing disciplines.

**FIFO** The simplest possible queueing discipline is FIFO (First-In-First-Out) that has only a single queue and a simple drop-tail dropper.

**PQ** PQ (Priority Queueing) has multiple queues associated with different priorities. A queue with a higher priority is always served first. Priority queueing is the simplest form of preferential queueing. However, low priority traffic easily starves unless there is a mechanism to regulate high priority traffic.

**WFQ** WFQ (Weighted Fair Queueing) [11, 4, 8] is a discipline that assigns an independent queue for each flow. WFQ can provide fair bandwidth allocation in times of congestion, and protects a flow from other flows. A weight can be assigned to each queue to give a different proportion of the network capacity.

**SFQ** SFQ (Stochastic Fairness Queueing) [10] is an approximation of WFQ. WFQ is difficult to implement because a large number of queues are required

as the number of flows increases. In SFQ, a hash function is used to map a flow to one of a fixed set of queues, and thus, it is possible for two different flows to be mapped into the same queue.

**CBQ** CBQ (Class Based Queueing) [7] achieves both partitioning and sharing of link bandwidth by hierarchically structured classes. Each class has its own queue and is assigned its share of bandwidth. CBQ is non-work conserving and can regulate bandwidth use of a class. A child class can be configured to borrow bandwidth from its parent class as long as excess bandwidth is available.

**RED** RED (Random Early Detection) [6, 2] is a dropper mechanism that exercises packet dropping stochastically according to the average queue length. RED avoids traffic synchronization in which many TCPs lose packets at one time [5]. Also, RED makes TCPs keep the queue length short. RED is fair in the sense that packets are dropped from flows with a probability proportional to their buffer occupation. Since RED does not require per-flow state, it is considered scalable and suitable for backbone routers.

### 1.3 Issues in Queueing

Although there are a large number of mechanisms available for traffic management, there is no single mechanism that satisfies a wide range of requirements. Therefore, it is important to use appropriate mechanisms suitable for a purpose.

In addition, a mechanism can have quite different effects depending on how it is used. For example, WFQ for best effort traffic can provide fair bandwidth allocation. A certain portion of the link capacity can be reserved by configuring the weight and the classifier of WFQ. Further, the delay can be bounded by adding a token bucket to the traffic source.

Furthermore, it is not easy to combine different mechanisms in a coherent manner because different mechanisms are independently developed to meet the requirements of specific applications.

The issues in designing queueing disciplines are described below.

**Overhead** Most of the functional blocks are located in the packet forwarding path, and thus, adds some overhead to forwarding performance. A queueing discipline should require only a few simple operations to forward a packet in order to scale to a high-speed network. It is also preferable to be easily implemented in hardware.

**Flow Definition** A flow is a unit that a classifier identifies packets in a traffic stream. A flow can be a micro flow such as a single TCP session, or some type of aggregated flow.

Packets belonging to the same micro flow should be placed in the same queue in order to avoid packet re-ordering. Although TCP and other transport mechanisms can handle out-of-order packets, frequent out-of-order packets will considerably damage the transport performance.

**Classifier Design** The design of an efficient classifier is still an area of active research. A scalable algorithm is required as the number of filters or the number of active flows increases. Efficient handling of wild card filters is difficult because it needs to find a best match for multiple fields.

Classifiers are required not only by queueing but also by firewall, layer 4 forwarding, and traffic monitoring. Classifiers should be designed to be shared by other components.

To identify traffic types by port numbers, a classifier needs to check the transport header (e.g., TCP, UDP). However, port based classification is not always possible if a packet is fragmented or encrypted. Although IP fragments will decrease by the Path MTU Discovery, encrypted packets will be common with the widespread use of secure shells and IPsec.

**Required States** A queueing discipline needs to keep some state for each traffic class. The size of a state and the total number of states have a great impact to the scalability of a queueing discipline. It is believed that per-flow queueing is preferable for a small network or at an edge of a backbone but only aggregated-flow queueing is possible within a backbone network.

A related issue is how long a flow state is maintained. A queueing discipline could keep only the states of flows that have packets in the queue. On the other hand, a discipline would need information for a longer period to enforce a longer term rule.

**Fairness** Fairness is an important property to handle best effort traffic. However, there are different definitions of fairness and different targets for whom fairness is defined. Local fairness at a router does not necessarily lead to global fairness. Besides, network traffic is dynamic and constantly changing so that static fairness does not necessarily lead to fairness in a larger time scale.

**Non-work Conserving Queues** A work-conserving discipline is idle only when there is no packet awaiting service. A non-work conserving discipline, on the other hand, can delay packets in the queue; it can be considered as a form of a shaper. A non-work conserving queue is more complex to implement but is able to limit the peak rate, reduce jitters, and provide tighter performance bounds.

**Statistical Guarantee** Performance guarantee can be either deterministic or statistic. In general, deterministic guarantee requires a much larger fraction of the resources to be reserved than statistical guarantee. In practice, deterministic guarantee is difficult to implement because computer communication involves many mechanisms that do not have tight bounds.

## 2 Traffic Management

There are people arguing that there are no need for QoS control since bandwidth will be cheap and abundant in the future. However, traffic management is not a choice between QoS and non-QoS but a wide range of spectrum. For example, at one extreme, every single packet could be precisely controlled at every router. At the other extreme, packets could be transferred even without flow control. However, both approaches are too expensive to realize and to manage so that they have no practical importance.

For a properly provisioned network, queue management could be considered as a precaution in case of congestion. It also works as a protective measure against misbehaving flows, misconfiguration, or misprovisioning. The effect of active queue management will not be so visible for such a properly provisioned network. However, it will virtually shift the starting point of congestion so that the effect is similar to increasing the link capacity.

Traffic management needs a good balance between controlling and provisioning at each level and among different levels. It is important to find a balance point that is cost-effective as well as administratively easy to manage.

### 2.1 Time Scale of Traffic Management

Traffic management consists of a diverse set of mechanisms and policies. Traffic management includes price-

ing, capacity planning, end-to-end flow control, packet scheduling, and other factors. These cover different time scales and complement one another.

The time scale of queueing is a packet transmission time. Queueing is effective to manage short bursts of packets. End-to-end flow control in turn manages the rate of a flow in a larger time scale. An important role of end-to-end flow control is to keep the size of packet bursts small enough to be manageable by queueing. To this end, large capacity itself is of no use for managing bursts in the packet level time scale. On the contrary, widening gap in link speed makes bursts larger and larger so that it makes managing traffic more important, especially at bandwidth gap points.

## 2.2 Controlling Bottleneck Link

Typically, bottleneck points are entries of WAN connections and they are the source of packet loss and delay. Queue management is most effective at those points.

Congestion is often caused by a small number of bulk data sessions (e.g., web images, ftp) so that isolating such sessions from other types of traffic will significantly improve network performance. It also serves as a protective measure. On the other hand, RED will substantially improve the performance of cooperative TCP sessions.

There are network administrators trying to keep the link utilization as high as possible. However, queueing theory tells us that the system performance drastically drops if the link utilization becomes close to 100%. It is a phenomenon that a queue is no longer able to absorb fluctuations in packet arrivals. Ideally, the link capacity should be provisioned so that the average link utilization is under a certain point, say 80%.

A difficulty in deploying queue management is that queueing manages only outgoing traffic and the beneficiaries are on the other side of a link. Queueing is not appropriate for managing incoming traffic because the queue is almost always empty at the exit of a bottleneck. In order to manage incoming traffic, queue management should be placed at the other end of the WAN link but most organizations do not have control over it.

## 2.3 Queueing Delay

Network engineers tend to focus on the forwarding performance. That is, how many packets can be forwarded per second, or how long it takes to forward a single packet. However, once the forwarding overhead becomes less than a packet transmission time, the throughput reaches the wire speed by a pipeline effect. Although further cutting down the overhead improves the delay, it has no effect if the queue is not empty.

On the other hand, queueing delay (waiting time in the queue) is by orders of magnitude larger than the forwarding delay. It implies that, if there is a bottleneck, high-

Table 1: Queueing Overhead Comparison

	FIFO	FIFOQ	RED	WFQ	CBQ	CBQ +RED
(usec)	0.0	0.14	1.62	1.95	10.72	11.97

speed forwarding does not improve the delay because most of the delay comes from queueing delay. Thus, we should pay closer attention to queueing delay, once the throughput reaches the wire speed.

## 2.4 Impact of Link Speed

It is important to understand how the effects and the overheads of queueing are related to the link speed. To illustrate the issues involved, Figure 2 plots packet transmission time and queueing delay on varying link speed in log-log scale. **min delay** and **packet delay** show the required time to transmit a packet at the wire speed with the packet size of 64 bytes and 1500 bytes, respectively. These are the minimum time required to forward a packet by a store-and-forward method. **worst delay** shows the worst case queueing delay when the queue is full, assuming that the maximum queue length is 50 (the default value in BSD UNIX) and all packets are 1500-byte long. On the other hand, Table 1 shows the per-packet overhead of different queueing disciplines measured on a PentiumPro 200MHz machine [3].

The per-packet overhead of queueing is independent of link speed. By a simplistic analysis, queueing overhead would be negligible if the per-packet overhead is less than **min delay**, and could be acceptable if the per-packet overhead is less than **packet delay**. The overhead of CBQ is 10usec. It would be negligible up to 40Mbps and acceptable even at 1Gbps. The overhead of RED is 1.6usec. It would be negligible up to 300Mbps.

On the other hand, the delay requirement of an application is also independent of link speed. If an interactive telnet session needs the latency to be less than 300 msec, preferential scheduling is required for link speed less than 1.5Mbps. If a voice stream needs the latency to be less than 30 msec, preferential scheduling is required for link speed less than 20Mbps.

Although there are other performance factors and the analysis is simplistic, it illustrates the effects of the link speed on queueing. In summary, queueing does not have significant overhead for commonly used link speed. Preferential scheduling improves interactive response on a slow link, and improves real-time traffic on a medium speed link.

## 2.5 Building Services

So far, we have looked at the behavior of a single router. An end-to-end service quality can be obtained by con-

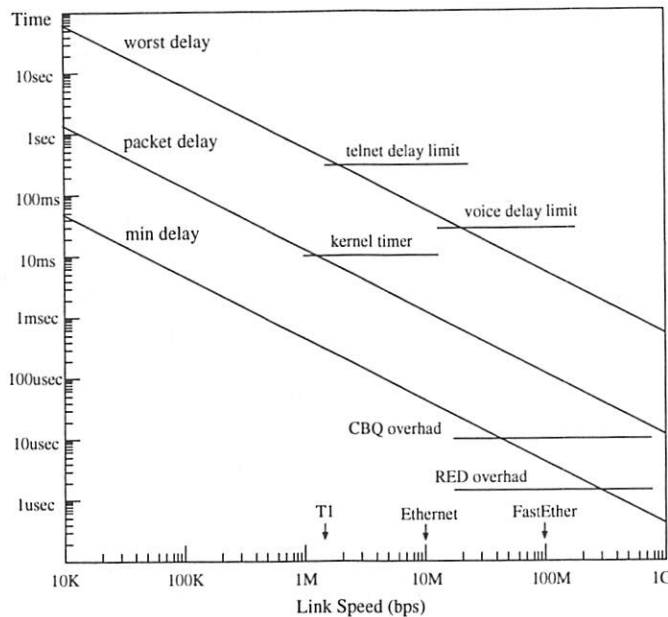


Figure 2: Queueing and Link Speed

catenating router behaviors along the communication path. For example, a traffic stream from user A to user B can be controlled such a way that the average rate is 1Mbps, the peak rate is 3Mbps and the packet delay is less than 1msec.

However, to make useful services, a network as a whole should be properly configured in a consistent way. In order to guarantee a service quality, it is necessary to configure all routers along the path and control all incoming traffic to these routers.

The diffserv working group at IETF is trying to establish a framework for various types of differentiated services [1]. In the diffserv model, a network that supports a common set of services is called "DS domain". A DS domain should be built in such a way that all incoming packets are policed at the boundary. Incoming packets are classified, measured and marked according to the user contract. These boundary actions are called "traffic conditioning". Inside a DS domain, internal routers (called DS interior nodes) perform preferential packet scheduling using only the packet header field (DS field) that has been marked at the boundary.

Traffic management mechanisms can be simpler in a closed network that can police all incoming traffic at the network boundary. For example, a simple priority queueing discipline can provide a premium service if the amount of incoming premium traffic is limited to a small fraction of the capacity. On the other hand, most current IP networks do not follow such a closed network model so that no firm assumption can be made about incoming

traffic.

### 3 ALTQ

ALTQ [3] is a framework for FreeBSD that introduces a variety of queueing disciplines. ALTQ provides a platform for traffic management related research. ALTQ also makes active queue management available for operational experience.

#### 3.1 Design

The basic design of ALTQ is quite simple; the queueing interface is designed as a switch to a set of queueing disciplines as shown in Figure 3. To implement ALTQ, several fields are added to *struct ifnet*. The added fields are a discipline type, a common state field, a pointer to a discipline specific state, and pointers to discipline specific enqueue/dequeue functions.

The implementation policy of ALTQ is to make minimal changes to the existing code. The current kernel, however, does not have queueing abstraction enough to implement various types of queueing disciplines. As a result, there are many parts of the kernel code that assume FIFO queueing and the *ifqueue* structure.

Especially, it is problematic that many drivers directly use the *ifqueue* structure, *if\_snd*, in the *ifnet* structure. These drivers must be modified but it is not easy to modify all the existing drivers. Therefore, we took an approach that allows both modified drivers and unmodified drivers to coexist so that we can modify only the drivers we need, and incrementally add supported drivers.

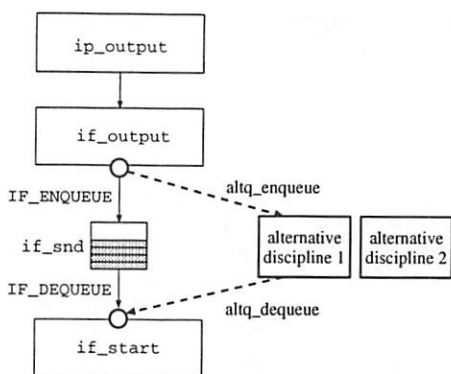


Figure 3: Alternate Queueing Architecture

### 3.1.1 Queueing Operations

In ALTQ, queueing disciplines have a common set of queue operations. Other parts of the kernel code manipulate a queue through 4 queueing operations; enqueue, dequeue, peek and flush. Drivers are modified to use only these operations, and not to refer to the *ifqueue* structure.

The enqueue operation is responsible not only for queueing a packet but also for other functions such as classifier and dropper that are required to enqueue a packet.

The dequeue operation returns the next packet to send. The main role of the dequeue operation is packet scheduling.

The peek operation is similar to the dequeue operation but it does not remove the packet from the queue. The peek operation can be used by a driver to see if there is enough buffer space or DMA descriptors for the next packet. ALTQ does not have a prepend operation since prepending a packet does not make sense if a discipline has multiple queues. Therefore, a driver should use a peek-and-dequeue policy if necessary.

The flush operation is used to empty the queue since non-work conserving queues cannot be emptied by a dequeue-loop.

### 3.1.2 Discipline Operations

Queueing disciplines are controlled by *ioctl* system calls via a queueing device (e.g., */dev/cbq*). ALTQ is defined as a character device and each queueing discipline is defined as a minor device of ALTQ.

There are 4 common operations to handle queueing disciplines; attach, detach, enable, and disable. The attach operation sets a queueing discipline to the specified interface. An interface can have one queueing discipline attached at a time. The attached discipline is not activated until the enable operation is performed. When

the alternative queueing is disabled or closed, the system falls back to the original FIFO queueing.

Other than these operations, each queueing discipline usually needs discipline specific settings, which are also done via discipline specific *ioctl*s.

## 3.2 Using ALTQ

ALTQ implements several queueing disciplines including CBQ, WFQ, RED, ECN, and RIO. For managing an operational network, CBQ will be the most appropriate discipline. CBQ is flexible to meet a wide range of requirements and the implementation is stable and well tested. Moreover, the CBQ implementation also integrates RED so that RED can be enabled for each CBQ class. The detailed mechanism of CBQ can be found elsewhere [7, 12, 3].

There are implementation issues when using ALTQ for different link speeds. These issues are described below.

### 3.2.1 Effect of Timer Resolution

Shapers are usually realized using timers, and thus, the resolution of a shaper is limited by the kernel timer resolution. The kernel timer resolution is 10msec in most UNIX systems. This implies that traffic becomes bursty if the packet transmission time of the link is less than the kernel timer resolution. On 10Mbps Ethernet, a 1500 byte packet takes 1.2msec so that 8 packets can be sent during a timer interval. 100Mbps FastEthernet is problematic since more than 80 packets can be sent during a timer interval. Therefore, it is desirable to use a higher resolution for the kernel timer. Current PCs seem to have little overhead even if the timer resolution is increased by a factor of 10.

CBQ employs a more elaborate scheme to limit bandwidth but it also has constraints from the timer resolution. In CBQ, an overlimit class is suspended until the state becomes underlimit again. A suspended class can be resumed from transmission complete interrupts but it relies on a timeout in case that the class is not resumed from interrupts. CBQ also limits the number of back-to-back packets by a variable *maxburst*. In the worst case scenario in which resuming is done only from timeouts, bandwidth of a class is limited by the timer resolution and *maxburst*. The default value of *maxburst* is 16; the value is selected to achieve 9.6Mbps on Ethernet with the default timer resolution. However, it is only 1/10 of the link capacity for FastEthernet. It is desirable to use a higher timer resolution for FastEthernet; 1 msec resolution achieves 96Mbps.

Note that the timer resolution affects only non-work conserving disciplines. Work-conserving disciplines do

not need timers since packets are sent from transmission complete interrupts.

### 3.2.2 Difference in Network Cards

Some cards generate interrupts every time a packet is transmitted, and some generate interrupts only when the buffer becomes empty. It is generally believed that a smart network card should reduce interrupts to alleviate CPU burden. However, a queueing discipline could have finer grained control with frequent interrupts; it is a trade-off between CPU control and CPU load. There is an interesting report that CBQ works much better with an old NE2000 card that interrupts a lot and has small buffers.

### 3.2.3 Device Buffers

There is a similar trade-off in setting the buffer size in a network card. When delay is a concern on a slow link, large buffers in network cards could adversely affect queueing. For example, if a network card for a 128Kbps link has a 16KB buffer, the buffer can hold 1 second worth of packets. The device buffer has an effect of inserting another FIFO queue beneath a queueing discipline. This problem is invisible under FIFO but it becomes apparent when preferential scheduling is used.

The transmission buffer size should be set to the minimum amount that is required to fill up the link. Although it is not easy to automatically detect the appropriate buffer size, it seems that many drivers set an excessive buffer size.

## 3.3 Availability

A public release of ALTQ for FreeBSD, the source code along with additional information, can be found at <http://www.csl.sony.co.jp/person/kjc/software.html>.

## 4 Related Work

### 4.1 Dummynet

Dummynet [9] is another popular mechanism available for FreeBSD to limit bandwidth. Dummynet is originally designed to emulate a link with varying bandwidth and delay, and realized as a set of 2-level shapers; the first level shaper enforces the bandwidth limit, and the second level shaper enforces the specified delay.

Dummynet has several advantages over ALTQ. Dummynet is implemented solely in the IP layer so that it is device independent and no modification is necessary to drivers. Because dummynet is a set of software shapers, dummynet can be used both on the input path and on the output path. In addition, the classifier of dummynet is integrated into *ipfw* (the firewall mechanism of FreeBSD)

so that it can be configured as part of firewall rules. Dummynet also works with the Ethernet bridging mechanism.

On the other hand, there are disadvantages. The shaper mechanism is realized solely by the kernel timer so that the shaper resolution is limited to the kernel timer resolution as described in Section 3.2.1. Although ALTQ shares the same limitation, ALTQ can take advantage of transmission complete interrupts. Dummynet currently does not have a mechanism for packet scheduling nor active buffer management. Dummynet does not work with the *fastforwarding* mechanism that bypasses the normal IP forwarding path.

In summary, dummynet is good for simple bandwidth limiting on moderate (Ethernet class) link speed, and it is easy to configure. There are great demands for bandwidth control that fall into this category.

### 4.2 Linux Traffic Control

Linux has a traffic control (TC) framework since version 2.1. The implemented queueing disciplines include CSZ, PQ, CBQ, RED and SFQ.

Linux TC is similar to ALTQ in a number of ways. The Linux TC framework has a switch of queueing disciplines and defines a set of queue operations. One minor difference found in the queue operations is that Linux TC defines "requeue" (prepend) instead of "peek". Linux TC employs a dequeue-and-requeue policy while ALTQ employs a peek-and-dequeue policy.

The architectural differences come from the kernel architecture. That is, Linux has a network device layer and its *sk\_buff* has rich fields.

Linux has a common network device layer that handles link type specific processing and acts as an upper half of a driver. Queueing is done within this device layer so that TC requires changes only in this layer.

In BSD UNIX, there is no common code path between the network layer and network device drivers. Operations are performed only through *struct ifnet*. As a result, the ALTQ support is scattered in *if\_output* and *if\_start*. Note that it is not only ALTQ but also BPF and ethernet bridging need supporting code in device drivers. In Linux, they are also supported in the common network device layer.

Linux's *sk\_buff* has many fields and have almost all information about a packet. A classifier can easily access network layer or transport layer information.

On the other hand, *mbuf* of BSD UNIX carries no information about a packet. Though this design is good for enforcing network stack layering, a classifier needs to extract information from a packet itself by parsing headers.

These architectural differences illustrate the difference in their design philosophy. The network code of BSD UNIX has been successful with this minimalist approach.

However, BSD UNIX might need to redesign the current abstraction in the future. An abstracted network device will make extensions easier and keep drivers simpler. There are other possible extensions to the interface level such as sub-interfaces for VLAN and virtual interfaces for multi-link. Also, various optimizations will be possible if packet information can be tagged to *mbuf*.

## 5 Conclusion

There are increasing demands and expectations for network traffic management. Although a variety of technologies are available, there is no single mechanism that satisfies a wide range of requirements. It is important to understand advantages and limitations of different mechanisms.

It is also important to understand the system bottleneck for different link speeds. With a network ranging from a slow modem to a high-speed fiber, the system bottleneck shifts one place to another. The requirements for the hardware and the system configuration also change.

The behaviors of single queueing disciplines are well understood but interaction of different mechanisms, especially in operational settings, needs more study and experience. We hope ALTQ will be of use to gain experience in the field.

## References

- [1] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An architecture for differentiated services. RFC 2475, Internet Engineering Task Force, December 1998.
- [2] B. Braden, D. Clark, J. Crowcroft, B. Davie, S. Deering, D. Estrin, S. Floyd, V. Jacobson, G. Minshall, C. Partridge, K. L. Peterson, S. Shenker Ramakrishnan, J. Wroclawski, and L. Zhang. Recommendations on queue management and congestion avoidance in the internet. RFC 2309, Internet Engineering Task Force, April 1998.
- [3] Kenjiro Cho. A Framework for Alternate Queueing: Towards Traffic Management by PC-UNIX Based Routers. In *USENIX 1998 Annual Technical Conference*, pages 247–258, June 1998.
- [4] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. In *Proceedings of SIGCOMM '89 Symposium*, pages 1–12, Austin, Texas, September 1989.
- [5] S. Floyd and V. Jacobson. Traffic phase effects in packet-switched gateways. *Computer Communication Review*, 21(2):26–42, April 1991.
- [6] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transaction on Networking*, 1(4):397–413, August 1993. Also available from <http://www.aciri.org/floyd/papers.html>.
- [7] Sally Floyd and Van Jacobson. Link-sharing and resource management models for packet networks. *IEEE/ACM Transactions on Networking*, 3(4), August 1995. Also available from <http://www.aciri.org/floyd/papers.html>.
- [8] Srinivasan Keshav. On the efficient implementation of fair queueing. *Internetworking: Research and Experience*, 2:157–173, September 1991.
- [9] Rizzo L. Dummynet: A simple approach to the evaluation of network protocols. *Computer Communication Review*, 27(1):31–41, April 1997. Also available from <http://www.iet.unipi.it/~luigi/>.
- [10] P. E. McKenney. Stochastic fairness queueing. In *Proceedings of INFOCOM*, San Francisco, California, June 1990.
- [11] John Nagle. On packet switches with infinite storage. *IEEE Trans. on Comm.*, 35(4), April 1987.
- [12] Ian Wakeman, Atanu Ghosh, Jon Crowcroft, Van Jacobson, and Sally Floyd. Implementing real-time packet forwarding policies using streams. In *Proceedings of USENIX '95*, pages 71–82, New Orleans, Louisiana, January 1995.

# Opening The Source Repository With Anonymous CVS

Charles D. Cranor

*AT&T Labs-Research*  
*Florham Park, New Jersey, USA*  
chuck@research.att.com

Theo de Raadt

*The OpenBSD Project*  
*Calgary, Alberta, Canada*  
deraadt@openbsd.org

## Abstract

Anonymous CVS is an advanced source file distribution mechanism we created to allow *open source* software projects to distribute source code and information about code to Internet users. Built on top of the Concurrent Versions System (CVS) revision control system, Anonymous CVS safely allows anonymous read-only access to a CVS source repository. Prior to the introduction of Anonymous CVS, access to a CVS repository had to be restricted to a select group of privileged software developers. The advantage of open source software is that it promotes reliability and quality by allowing independent peer review and rapid evolution of source code. By introducing Anonymous CVS, we have extended the concept of open source software projects to *open source repository* projects. Having an open source repository allows users to take a more active role in the debugging and development of open source projects. In this paper we will examine and compare the mechanisms used by open source projects to distribute source code. We will present the design and implementation of the first Anonymous CVS server (used to distribute the OpenBSD operating system). We will explain some of our concerns (e.g., security) and some of the problems we faced when trying to adapt CVS for anonymous use. We also will present other more recent source file distribution mechanisms that make use of an open CVS repository. Anonymous CVS is currently being used by a number of projects including OpenBSD, FreeBSD, Mozilla, Ecgs, Gnome, Python, and GNUstep.

## 1 Introduction

Over the past few years open source software has made significant inroads in the mainstream software world [7]. The popularity of open source operating systems such as Linux [12] and BSD [5] has generated great interest in the open source development software model. The key

attributes of the open source model are that the source code for a software project is freely available (usually over the Internet) and that the code's license guarantees the right to read, redistribute, modify, and use the code freely. The advantage of open source software over closed proprietary software is that it promotes software reliability and quality by supporting independent peer review and rapid evolution of source code. Anyone on the Internet can download, examine, enhance, or debug an open source program. This enables an open source project to have a large Internet-based international developer community that is constantly working on improving the project.

While all users benefit from the open source model, only a relatively few users take advantage of having access to the source code. In fact, most users of open source programs install pre-compiled versions of programs from CDROM distributions or the Internet and never bother to either download, inspect, or modify the source code. The few users who do deal directly with the source code are usually open source developers. These developers have special needs that are only partly met by projects that fit the standard definition of "open source" [7]. For example, in addition to having access to current snapshot of a project's source code, it is also useful to have access to older versions of source files, annotated per-file revision logs (GNU-style "ChangeLog" files are a poor substitute for this), and the ability to set the files in a source tree to a specific date or release. It is also useful to be able to update a source tree to the latest version without having to download the entire tree while preserving local changes. Historically, revision control systems such as the Source Code Control System (SCCS) [9] and the Revision Control System (RCS) [11] have provided some of these features on a local basis.

RCS and SCCS were designed to manage small-scale projects with a centralized set of developers thus they are not well-suited for large Internet-based open source

software projects. Neither system has an Internet server function that allows developers to check out a working copy of a source tree on their local systems, make modifications to it, and then merge those changes back into the main repository. The introduction of the RCS-based Concurrent Versions System (CVS) [1] revision control system addresses these issues. It allows large source trees to be managed as a group under RCS, and it has a network server mode that allows developers to be distributed across the Internet and yet share the same CVS source repository.

Prior to our work on Anonymous CVS, in order to be able to use CVS to access a source repository one had to have an account on the machine hosting the repository. Furthermore, the account had to have write access to the RCS files in the repository. Thus, open source projects that used CVS to manage their source trees had to restrict access to their repositories to a select group of privileged software developers in order to protect themselves from malicious attacks on their source tree. An unfortunate side effect of this was non-privileged users and developers could not access the CVS-based source tree and thus were locked out from the information contained in it. Denying users access to this information runs counter to the open source philosophy and reduces the effectiveness of the open source development model by making it more difficult for non-privileged users to download, debug, and manage their source trees.

In the Fall of 1995 when we started our own open source operating system project called OpenBSD, we decided to use CVS to manage the OpenBSD source tree. Based on our experiences with the previous open source project we were involved with, we recognized the inherent conflict between trying to maintain an open environment while maintaining a private CVS source repository that only privileged users could access. To resolve this conflict we created Anonymous CVS — a mechanism that lets anonymous Internet users access a source repository without compromising its security. Anonymous CVS evolves the open source concept to the next level: *open source repository*. The advantage of open source repository projects over plain open source projects is that it puts the information and power contained in a CVS-based source repository in the hands of the average developer. With Anonymous CVS the revisions, histories, and branches of a CVS tree are public. Anonymous CVS makes it easy to keep a large source tree up to date, even over a slow-speed modem link. The OpenBSD project even ships its CDROM with a checked out CVS tree so that OpenBSD users who are interested in using CVS can start right away without having to download the whole tree from scratch. Anonymous CVS also acts as a training ground by allowing developers new to the project and/or CVS to safely get experience

with the `cvcs` command before being given write access to the repository. We believe that once developers have had experience with an open source repository project they will find the development environment offered by a plain open source project to be inadequate.

In this paper we examine the issue of source code distribution for open source projects. In Section 2 we examine non-CVS based distribution mechanisms used (for the most part) prior to the introduction of Anonymous CVS. In Section 3 we describe the design and implementation the first Anonymous CVS servers, including issues such as CVS limitations, file locking, and security. In Section 4 we present other CVS-based open source repository distribution tools that were introduced after introduction of Anonymous CVS. Finally, in Section 5 we close by providing pointers to the source code of the currently available open source repository distribution tools and also a list of open source repository projects and their respective CVS servers.

## 2 Traditional Distribution Mechanisms

Traditionally, open source projects have distributed their source code through a number of non-CVS based mechanisms including USENET `comp.sources` newsgroups, anonymous FTP, web, and SUP. Recently projects have also started using Rsync and CTM for source distribution. While each of these mechanisms are useful for distributing code, they do not address the issue of distributing the types of meta information available in a CVS source repository. In this section we examine each of these mechanisms in more detail.

In the 1980s and early 1990s the moderated USENET `comp.sources` newsgroups were a popular way to distribute open source code. To submit a program, an author e-mailed the source code to the moderator of the appropriate USENET group. The moderator would then compile and test the code, and if the program functioned properly post it to the newsgroup as a series of articles. As the postings worked their way through the network, users would collect them and unpack, compile, and install the program. As the program evolved, the author of the program could forward patches to the moderator to test and post to the newsgroup. There are several problems that make these USENET newsgroups a less than ideal forum for the distribution of open source code. First, the group moderator is a bottleneck. Postings can be delayed weeks or even months awaiting the moderators attention. This does not mesh well with the rapid development environment associated with open source projects. While it is possible to have an unmoderated source newsgroup, it is not practical due to abundance of non-uniform and non-source postings (e.g., see `alt.sources`). Second, moderating an active source

newsgroup is hard work and it is difficult to find volunteers to perform this thankless task. Third, handling multipart source postings is irritating for users since they must collect all the parts (tracking down parts that are missing) and then assemble them together. Given the abundance of Internet connectivity, it is often easier to just FTP the sources rather than try and collect them from USENET. Thus, it is not surprising that the USENET source newsgroups are now mostly inactive.

Anonymous FTP and web servers are popular ways to distribute both binary and source code. Archiving a collection of tar, zip, or RPM files containing snapshots of a project's source code allows users to conveniently access programs on demand through the Internet. Web servers have the additional advantage of being able to include explanatory information intermixed with hypertext links to source distribution files. There are a number of disadvantages to this type of distribution mechanism. First, it forces developers to break their distribution up into periodic releases. If there is a large amount of time between releases then there is a large delay between when changes are made and when they get distributed to developers on the Internet. If the amount of time between releases is short, then the FTP or web site becomes crowded with numerous release archive files, patch files, or both. If the distribution is large, then downloading new releases becomes painful for developers who are attached to the Internet via slow modem links because in order to stay current new releases must be constantly downloaded. If patch files are used, then developers have the added overhead of downloading and applying the patches. Finally, old releases are often removed from the FTP or web server in order to conserve disk space. This makes it difficult to retrieve and compare old versions of a distribution with new versions. The Linux kernel and GNU programs have traditionally been distributed through these mechanisms.

Another way to distribute code is through a Software Upgrade Protocol (SUP) server [10]. SUP servers operate by tracking the modification times of a collection of source files. SUP clients track the time they were last run successfully. When a SUP client is run it connects to a server and asks for files that have changed since the last successful run. The SUP server checks its timestamp database and delivers only those files. The SUP server can run the files through a compression program to reduce the bandwidth required to update a source tree. The advantage of SUP is that only the files that have changed are downloaded. The disadvantage of SUP is that local changes to source files are not preserved and entire files must be downloaded when they are changed. Also, SUP does not supply any revision information or allow older versions of files to be accessed. Both CMU and the BSD open source operating systems projects have made ex-

tensive use of SUP to distribute source files.

The Rsync distribution program performs a similar function to SUP, but in a more efficient way [13]. In SUP when a file is updated the entire file is transferred, however in Rsync only the changes are sent. Rather than using a timestamp database, Rsync simply compares the timestamps and sizes of the source and target versions of a file. If there is a match then the file is not transferred. On the other hand, if the file does not match, then Rsync performs a rolling checksum over the file to determine where changes have been made. Rsync uses the results of this checksum to generate the differences between the source and target versions of the file. The advantage of this approach is that it has lower bandwidth requirements because only the changes are transferred, and the rolling checksum algorithm eliminates the need to have both versions of a file on the server in order to generate a diff. However, Rsync still has the limitation that it does not preserve local modifications to source files, and it does not provide access to older versions of source files or access to the types of meta information stored in a CVS repository.

CTM ("Current Through e-Mail") is another software distribution mechanism that transfers only the changes made to a collection of files rather than entire files [4]. CTM was designed to use the electronic mail as a data transport mechanism. CTM operates by comparing an old and new source tree and generating the differences between them. The diffs are broken up into e-mail sized chunks and mailed to a mailing list. CTM users collect the diffs from the mailing list and apply them to their local source tree by using the CTM client program. The main advantage of CTM is that it does not require IP connectivity in order to use, but compared to CVS-based mechanisms it is still limited.

### 3 Anonymous CVS Design and Implementation

CVS can be used to manage the source files of a source tree. The source files are stored as a collection of RCS control files called the CVS repository. Developers check out working, fully writable versions of a source tree, make modifications to the files, and check the changes back into the repository. CVS can also merge in changes committed by other users into a local repository, display commit log messages, check out specific branches or dated versions of a source tree, annotate each line of a source file with the revision and author of that line, and update a source tree by transmitting only a compressed version of the changes made to a file. Thus, CVS provides a more powerful and useful abstraction for open source developers than any of the software dis-

tribution mechanisms described in the previous section. However, prior to the introduction of Anonymous CVS, CVS had a major limitation for open source projects: an account with write access to the CVS source repository was required in order to use CVS. One of our goals in creating Anonymous CVS was to allow greater access to the OpenBSD project's CVS source repository in order to have a more open project and to encourage developer interest. We wanted to allow anyone on the Internet to safely have anonymous<sup>1</sup> read-only access to our CVS repository — a practice that was unheard of at the time.

### 3.1 Anonymous CVS Goals

As we were designing OpenBSD's Anonymous CVS service, we had the following three goals in mind:

**Security:** While we wanted to allow the world to have read access to our CVS repository, we did not wish to allow anonymous *write* access to it. Thus we had to ensure that our Anonymous CVS system did not compromise the security of our source repository.

**Efficiency:** CVS server operations are known for being resource intensive. While we were eager to provide anonymous access to our repository, we did not want to do so at the expense of bogging down our CVS machine. Thus we had to ensure that Anonymous CVS did not place an undue burden on our CVS system.

**Convenience:** If an Anonymous CVS service is difficult to access then no one will use it. Thus we designed our Anonymous CVS system to be as easy and convenient to use as possible. For systems with CVS installed, accessing our CVS repository is as easy as setting an environment variable and running CVS. No usernames, passwords, or special programs (other than CVS itself) are required to use Anonymous CVS.

### 3.2 Anonymous CVS Design

Based on our three goals we decided that Anonymous CVS service should be offered from a machine other than our main CVS server system. This provides security by keeping all anonymous connections off the main CVS server. The main server need only distribute a copy of its CVS-controlled RCS files to the anonymous system using a standard technique such as SUP. It does not have to trust the anonymous server

<sup>1</sup>By "anonymous" we mean that resources can be accessed without authenticating the user (as in anonymous FTP). Achieving truly anonymous access is a more difficult problem that was beyond the scope of our needs. A more anonymous access mechanism could be achieved by borrowing ideas from a system such as Crowds [8].

system beyond that. This also provides efficiency by keeping Anonymous CVS server and networking load off the main server machine. This is important because our main CVS server (`cvs.openbsd.org`) is connected to the Internet by a low-bandwidth ISDN link. Figure 1 shows the relationship between the main OpenBSD CVS server and the primary OpenBSD Anonymous CVS server (`anoncvs.openbsd.org`). To access the Anonymous CVS service, users simply set their `CVSROOT` environment variable to `anoncvs@anoncvs.openbsd.org` and run CVS commands normally. The Anonymous CVS server will reject any attempt to modify its local copy of the CVS repository.

We also secured the environment on the Anonymous CVS server in order to prevent malicious tampering with CVS service. Anonymous CVS is accessed through the special account "anoncvs." While this account has no password (thus allowing anyone to log into it), it also has a special anoncvs shell that restricts what it can run to a single command: "cvs server." Any attempt to run a command other than the CVS server results in the anoncvs shell printing an error message and exiting. When the anoncvs shell receives a request to run the CVS server it uses the `chroot` system call to restrict access to the server to a sandboxed environment. In order to use `chroot` the anoncvs shell must be setuid "root." While this is not optimal, we note that the anoncvs shell is a small program that immediately drops privileges as soon as it uses `chroot`. We feel that the gains of using a restricted root environment are worth the risks of having a small setuid program. A partial listing of the anoncvs shell is shown in Figure 2.

The only files that reside in the `chroot` environment are the commands necessary to run CVS (the `cvs` binary and helper RCS commands) and the read-only copy of the CVS repository from the main CVS server. Note that CVS version 1.10 and later versions access RCS files directly and thus the RCS helper commands no longer need to be in the sandbox area. The copy of the CVS repository in the sandbox is owned by a user other than the anoncvs user to prevent any chance of an anonymous user writing to it. The only writable directory in the sandbox environment is `/tmp` which is required for CVS to operate properly. Also note that there are no setuid files in the sandbox. Thus, if an anonymous user was to break out of the CVS server (e.g., through a buffer overflow) it would be very difficult to do any damage other than interfere with other CVS server processes running under the anoncvs account. The possibility of such interference could be avoided by allowing the anoncvs shell to randomly distribute its UID among a specific range of UIDs reserved for anonymous access.

The main advantage of using an anoncvs shell rather

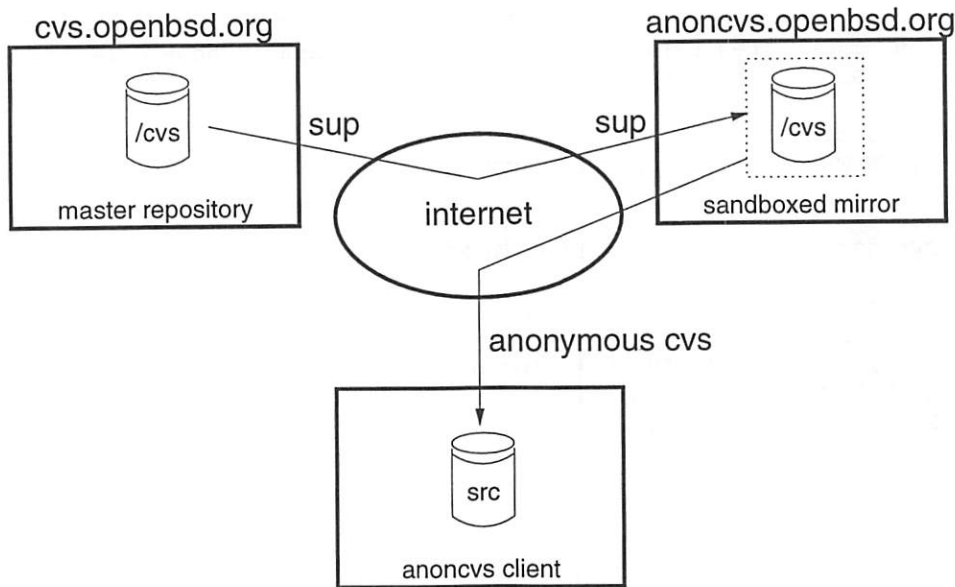


Figure 1: OpenBSD's Anonymous CVS service. The CVS repository is mirrored within the sandbox using SUP (Rsync could also be used).

than a specialized server program is that it integrates nicely with CVS's server system and can be used with standard login programs such as rsh and ssh. Many Anonymous CVS servers disallow rsh access and allow only ssh access for added security. For ssh, a non-standard server port such as 2022 can be used in addition to the standard port to make Anonymous CVS more firewall friendly. This allows users to work around poorly thought-out firewalls that are configured to block all unknown traffic in the reserved TCP port range.

Note that an Anonymous CVS server is more secure than most Anonymous FTP servers. This is because standard FTP servers never revoke their root access. Instead, they just swap UIDs when accessing files. On the other hand, the anoncvs shell permanently revokes root access before running the CVS server.

### 3.3 Anonymous CVS Implementation Issues

As we implemented Anonymous CVS, we encountered three issues relating to CVS that caused us some concern. First, we discovered that CVS did not run properly without write access to its log file. Since CVS itself is an open source program, we fixed this problem by adding the CVSREADONLYFS environment variable to CVS. If set, CVS ignores this error.

Our second concern with Anonymous CVS was how it interacted with CVS's file locking protocol. CVS controls access to RCS files by creating lock files in the CVS

repository area. Under Anonymous CVS this is not possible since the Anonymous CVS server runs under a UID that does not have write access to the repository. To address this issue we disabled file locking for read-only access. Since commits are not allowed in the Anonymous CVS server's copy of the repository this is not a problem. However, another possible problem is that the Anonymous CVS server may encounter a partially complete RCS file in its copy of the repository. We examined the CVS documentation and source code and determined this was unlikely for the following two reasons. First, CVS on the main server updates its RCS files in one operation by creating a temporary RCS file, modifying it, and finally renaming it to the RCS file. Since the rename system call is atomic, there is no chance of the mechanism used to transfer RCS files from the main server to the anonymous server encountering an incomplete RCS file. Second, we use SUP to transfer our RCS files from the main server to the anonymous server. When SUP installs an updated file it uses the same atomic-rename technique that the CVS server uses to install a new file. This prevents the CVS servers running on the anonymous server from seeing an incomplete RCS file. One possible problem that could be encountered is if CVS reads a list of RCS files currently in the mirrored repository and SUP deletes one of those RCS files before CVS has a chance to open it. In reality the odds of this happening are very low because RCS files typically do not get removed from a repository. Of course when multiple files are being updated on the mas-

```

/* location of CVS tree relative to anonymous CVS user's home directory */
#define LOCALROOT      "/cvs"

/* remote hostname */
#define HOSTNAME        "anoncvs@anoncvs1.usa.openbsd.org"

/* cvs root */
#define CVSROOT          __CONCAT3(HOSTNAME, ":", LOCALROOT)

/* default environment */
char * const env[] = {
    "PATH=/bin:/usr/bin", "SHELL=/bin/sh",
    __CONCAT3("CVSROOT=", LOCALROOT),
    "HOME=/", "CVSREADONLYFS=1",
    NULL
};

int main(argc, argv)

int argc;
char *argv[];

{
    struct passwd *pw;

    pw = getpwuid(getuid());
    if (pw == NULL || pw->pw_dir == NULL)
        errx(1, "no user/dir for uid 0", getuid());

    setuid(0);
    if (chroot(pw->pw_dir) == -1)
        errx(1, "chroot");
    chdir("/");
    setuid(pw->pw_uid);

    /* program now "safe" in sandbox with root privs dropped */
    if (argc != 3 || strcmp("anoncvssh", argv[0]) != 0 ||
        strcmp("-c", argv[1]) != 0 || (strcmp("cvs server", argv[2]) != 0 &&
        strcmp(__CONCAT3("cvs -d ", LOCALROOT, " server"), argv[2]) != 0)) {
        fprintf(stderr, "\nTo use anonymous CVS install the latest ");
        fprintf(stderr, "version of CVS on your local machine.\n");
        fprintf(stderr, "Then set your CVSROOT environment variable ");
        fprintf(stderr, "to the following value:\n");
        fprintf(stderr, "\t\t\t\t\t", CVSROOT);
        sleep(10);
        exit(0);
    }
    execle("/usr/bin/cvs", "cvs", "server", NULL, env);
    perror("execle: cvs");
    fprintf(stderr, "unable to exec CVS server!\n");
    exit(1);
}

```

Figure 2: Partial listing of anoncvs shell program

ter CVS server there is still a chance that an anonymous user will end up fetching some old file and some new files from the group of files being updated. Indeed, this is even a problem for normal CVS users because updates are often checked in multiple chunks. However, in practice we have found that this problem does not occur that often. In the future it may be useful to determine ways to extend parts of CVS's file locking to anonymous servers. It would also be useful to create a mechanism where CVS updates on the master server are pushed to the slave anonymous servers as soon as they happen.

The third issue relating to CVS that caused us concern was CVS's poor handling of network flow control. CVS's server function was designed to run in a high-bandwidth network environment with a relatively small source repository. This environment is fundamentally incompatible with our target environment. The OpenBSD source tree consists of 250MB of source files, and we distribute it to many anonymous users connected to the Internet via slow speed PPP links. We found that CVS did not run well in this environment because it was designed to minimize the amount of time it holds a lock on a directory in a repository. In order to do this, when checking out source code the CVS server splits into two processes. The first process walks the CVS repository's directory tree as fast as possible performing the requested action. The second process buffers the output from the first process in its memory and sends it out over the network connection. The second process uses non-blocking I/O to ensure that it does not block on a slow network connection. This allows the first process to run to completion without blocking on full network I/O buffers while holding a lock on a repository directory. The problem with this design is that the CVS developers did not put a limit on the amount of data the second process was willing to buffer. The result of this is that for a large checkout over a slow link the second process can grow and consume large chunks of virtual memory. We found that if multiple Anonymous CVS servers were running at the same time they quite often exhausted all available virtual memory on our Anonymous CVS server machine thus creating a denial of service. This problem was especially annoying since locking is not an issue with a read-only CVS repository.

To fix this problem, we modified CVS to limit the amount of data the second process can buffer. In our environment it is better to let the first process block than to run our server out of virtual memory. Partly due to our complaints about the behavior of CVS in this case, the maintainers of CVS modified it to address this issue. Their fix was to modify the first process in a CVS checkout to be non-blocking only on a per-directory basis. This allows the second process to catch up to the first after the first has completed a directory. The ad-

vantage of this fix is that it minimizes the time a CVS directory lock can be held. There is still potential for problems if CVS encounters a single directory with a large number of modified files. In this case it is still possible for the CVS server to use a significant chunks of system virtual memory. However, as most source files in large sources trees are distributed among several directories this should not be a problem.

One remaining unsolved issue is the fact that CVS requires a writable `/tmp` directory in order to function. For better security we would like for an Anonymous CVS server to be able to function without any write access to the filesystem in the `chroot` environment in which it operates.

## 4 Other CVS-based Distribution Mechanisms

As open source repository projects became more widespread, several new tools including CVS's Pserver, CVSWeb, and CVSup were developed to take advantage of this powerful new environment.

CVS's Pserver was created by the CVS development team partly in response to the demand for anonymous support within CVS itself. Rather than use the standard CVS server with the `anoncvs` shell that we created, CVS's Pserver bypasses `rsh/ssh` and listens on its own TCP port for connections. Pserver's user interface requires the use of a login and password (even for anonymous access – an annoying inconvenience for users getting started with Anonymous CVS), and it transmits this data over the wire in clear-text. Pserver often does not operate in a `chroot` environment, and thus it is more of a security risk than our version of Anonymous CVS. It is possible to run Pserver in a `chroot` environment, but it requires more files to be added to the sandbox environment in order for Pserver to authenticate the user, especially on systems that support complex user authentication mechanisms like PAM [3]. Pserver, unlike Anonymous CVS, also does not fully give up root privileges if it has them. In the context of anonymous access, the main advantage of Pserver is that it is included with the main CVS distribution.

The CVSWeb system was developed by Bill Fenner of the FreeBSD project to allow anonymous access to a CVS repository through a standard web browser [2]. Although CVSWeb cannot be used in the same way as Anonymous CVS to update a local source tree, the big advantage of CVSWeb is that it allows anyone with a web browser to easily browse the content of a CVS repository using a graphical user interface. This can often be more convenient than using the standard CVS interface.

Tool	Location
SUP	<a href="ftp://ftp.openbsd.org/pub/OpenBSD/src/usr.bin/sup">ftp://ftp.openbsd.org/pub/OpenBSD/src/usr.bin/sup</a>
Rsync	<a href="http://samba.anu.edu.au/rsync/">http://samba.anu.edu.au/rsync/</a>
CTM	<a href="http://www.freebsd.org/handbook/synching.html#CTM">http://www.freebsd.org/handbook/synching.html#CTM</a>
CVS (includes Pserver)	<a href="ftp://ftp.gnu.org/gnu/cvs">ftp://ftp.gnu.org/gnu/cvs</a>
Anonymous CVS	<a href="http://www.openbsd.org/anoncvshar">http://www.openbsd.org/anoncvshar</a>
CVSWeb	<a href="http://www.freebsd.org/~fenner/cvsweb/">http://www.freebsd.org/~fenner/cvsweb/</a>
CVSup	<a href="http://www.polstra.com/projects/freeware/CVSup/">http://www.polstra.com/projects/freeware/CVSup/</a>

Table 1: Source distribution tools

Project	Information Pointer
CMU Common Lisp	<a href="http://www3.cons.org/cmuc1/">http://www3.cons.org/cmuc1/</a>
Ecgs	<a href="http://egcs.cygnum.com/cvs.html">http://egcs.cygnum.com/cvs.html</a>
FreeBSD	<a href="http://www.freebsd.org/handbook/synching.html#ANONCVS">http://www.freebsd.org/handbook/synching.html#ANONCVS</a>
Gnome	<a href="http://www.tw.gnome.org/software/anoncvsh.html">http://www.tw.gnome.org/software/anoncvsh.html</a>
GNUstep	<a href="http://www.gnustep.org/resources/Anoncvsh.txt">http://www.gnustep.org/resources/Anoncvsh.txt</a>
Guile	<a href="http://www.red-bean.com/guile/guile-anon-cvs.html">http://www.red-bean.com/guile/guile-anon-cvs.html</a>
Mozilla	<a href="http://www.mozilla.org/cvs.html">http://www.mozilla.org/cvs.html</a>
Obtuse	<a href="http://www.obtuse.com/open_source/">http://www.obtuse.com/open_source/</a>
OpenBSD	<a href="http://www.openbsd.org/anoncvsh.html">http://www.openbsd.org/anoncvsh.html</a>
OpenLDAP	<a href="http://www.openldap.org/software/repo.html">http://www.openldap.org/software/repo.html</a>
Python	<a href="http://www.python.org/download/cvs.html">http://www.python.org/download/cvs.html</a>
Quinn Diff	<a href="http://quinn-diff.nocrew.org/anoncvsh.html">http://quinn-diff.nocrew.org/anoncvsh.html</a>
Sudo	<a href="http://www.courtesan.com/sudo/anoncvsh.html">http://www.courtesan.com/sudo/anoncvsh.html</a>

Table 2: Open source repository projects on the Internet

The current state of the art in open source repository source distribution tools is John Polstra's CVSup package [6]. CVSup is an efficient and flexible file distribution system. CVSup's efficiency is due to two factors. First, the control protocol used by CVSup streams multiple requests between client and server rather than making the client wait for a request to be satisfied before issuing the next request. This helps CVSup make the most of available network bandwidth. Second, CVSup takes advantage of knowledge of the internal formats of certain types of files to reduce the overhead of sending an update. CVSup knows the format of RCS files, CVS repositories, and append-only log files. CVSup can use this knowledge to easily extract the minimal amount of data necessary to send changes from these types of files over the network (the data can optionally be compressed before being transmitted). For files whose format CVSup does not understand, CVSup uses the Rsync algorithm. CVSup includes both a command line and GUI interface.

CVSup has two features that are especially useful for accessing CVS repositories. First, the CVSup client program can be used to request a specific version of a source tree. The version can be specified by date or by symbolic

name. Second, CVSup can be used to download changes from a master CVS repository and merge them directly into a local CVS repository<sup>2</sup>. This allows developers to maintain their local changes within a private branch of their copy of the master CVS repository. In order to achieve the same effect with traditional Anonymous CVS, one would have to go through the time consuming process of checking out a clean version of the master source tree (via Anonymous CVS) and then importing it into the vendor branch of a local CVS repository. CVSup can do the same job with much less overhead.

There are two drawbacks to CVSup. First, it is difficult to compile and install because it is written in Modula3 rather than C. While there are open source Modula3 environments available, compiling and installing them is a difficult task (especially for unsupported platforms). However, there are precompiled binaries available from the PostgreSQL project<sup>3</sup>. The second drawback of CVSup is that it can only access a set of pre-determined collections of files, while Anonymous CVS can access anywhere from a single file to the entire source tree.

<sup>2</sup>Care must be taken to avoid version number conflicts and deleted RCS files, see the CVSup FAQ for details.

<sup>3</sup>See [pub/CVSuponftp.postgresql.org](http://pub/CVSuponftp.postgresql.org)

However, if CVSup is used to download a copy of the entire repository, then standard CVS can be used on that repository to access individual files in that repository.

## 5 Conclusions

In this paper we have examined the issue of distributing the source code of open source projects to Internet developers. Table 1 contains a list of the tools discussed in this paper and pointers to where to get them. We examined the evolution of open source code distribution from early channels such as USENET and anonymous FTP to modern mechanisms such as Anonymous CVS, CVSWeb, and CVSup. Our contribution was the design and implementation of Anonymous CVS. Since the introduction of OpenBSD's Anonymous CVS service many other open source projects have opened up their CVS repositories. Table 2 contains URLs for some of the open source repository projects currently on the Internet. We believe Anonymous CVS has made a significant positive impact in the open source community. Anonymous CVS certainly had a positive impact on OpenBSD. We currently see around 2000 anoncvs transactions per-week on our Canadian-based Anonymous CVS server. Several of our other Anonymous CVS servers report similar usage. We hope to see more open source repository tools and projects appear on the Internet in the future.

## References

- [1] B. Berliner. CVS II: Parallelizing software development. In *USENIX Conference Proceedings*, pages 341–352. USENIX, 1990.
- [2] B. Fenner. CVSWeb. See [www.freebsd.org/~fenner/cvsweb.html](http://www.freebsd.org/~fenner/cvsweb.html).
- [3] Open Group. X/Open single sign-on service (XSSO) - pluggable authentication. See [www.opengroup.org/pubs/catalog/p702.htm](http://www.opengroup.org/pubs/catalog/p702.htm).
- [4] P. Kamp. Current through e-mail (CTM). See [www.freebsd.org/handbook/synching.html#CTM](http://www.freebsd.org/handbook/synching.html#CTM).
- [5] M. McKusick, K. Bostic, M. Karels, and J. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley, 1996.
- [6] J. Polstra. CVSup. See [www.polstra.com/projects/freeware/CVSup/](http://www.polstra.com/projects/freeware/CVSup/).
- [7] E. Raymond. Open source home page. See [www.opensource.org](http://www.opensource.org).
- [8] M. Reiter and A. Rubin. Anonymity loves company: Anonymous web transactions with crowds. *Communications of the ACM*, 42(2):32–48, February 1999.
- [9] M. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, SE-1(4):364–370, December 1975.
- [10] S. Shafer. The sup software upgrade protocol. Technical report, Department of Computer Science, Carnegie Mellon University, 1985.
- [11] W. Tichy. RCS – a system for version control. *Software – Practice & Experience*, 15(7):637–654, July 1985.
- [12] L. Torvalds et al. The Linux operating system. See [www.linux.org](http://www.linux.org).
- [13] A. Tridgell and P. Mackerras. The rsync algorithm. Technical report, Department of Computer Science, Australian National University, 1998.



## 1999 FREENIX Track Presentation

### Open Software in a Commercial Operating System

#### History and Topical Focus

Apple Computer has been working for the past couple of years to build a new and modern operating system to advance the Macintosh platform. The Macintosh platform, developed at Apple, while strong in many areas (in particular human interface), lacks the solid and modern foundation best exemplified in current variants of the Unix operating system. Apple acquired NeXT Software, Inc. in early 1997 largely in order to gain NeXT's experience in making an operating system based on a modern foundation which also had made advances in user interfaces, programming models, enterprise software, and other areas in which Apple was not traditionally strong. Thus began the development of Mac OS X Server, Apple's server operating system offering, which is also the basis for parts of Mac OS X, which we will be rolling out later in 1999.

This session will discuss this new foundation: the "Core Operating System", which is largely based on software which was developed at the University of California at Berkeley and Carnegie-Mellon University, and was made freely available with open licensing terms. In addition much of the Core OS utilizes software available from the present-day BSD efforts, the Free Software Foundation, M.I.T., the Apache Group, and countless other groups and individuals who create and publish software under open licenses.

#### The Value of Open Software

Creating an operating system is an enormous task. The quantity of code required to create a functional and useful system is quite large, and the nature of operating systems is that the quality of that code must be significantly higher than that of most software. Everything depends on the OS: Too little functionality make life difficult for the developers who wish to support your system. The smallest of bugs can disable most or all of the software than runs on the system--or worse, can cause crashes or data loss. It is therefore quite difficult indeed to create a whole operating system from scratch, and is a task not often undertaken.

It is fortunate, then, that there is a large body of software freely available which we have been able to use as a starting point. Even better, much open software is *good stuff*; there have been many years of development behind many of the efforts from which we have been able to draw our code.

The Mac OS X Server Core OS is a 4.4 BSD system, based on the 4.4 Lite2 release from Berkeley and updates with code from FreeBSD, NetBSD, and OpenBSD. BSD has seen many years of active development; it is well designed and has provided the basis for many established standards. It has proven to be a tried-and-true system and remains in the forefront as a platform in which to test and integrate emerging technology and ideas. It has become well-accepted that free operating systems are not only leading-edge experimental systems, but highly robust and scalable as well.

A non-trivial advantage of open software is the excellent developer support behind many open software projects. Apple has considerable experience with operating systems (we've shipped Mac OS, A/UX, AIX, and Linux systems, and worked on several others), but the engineering effort required to simply maintain and update in-house software is considerable and expensive.

At Apple, I write and maintain software, manage the source code control for all of Core OS (using open

software--CVS), go to lots of meetings, and oversee internal releases. In addition, I personally maintain over 60 open software projects in the Core OS build cycle, several of which are quite large. While these 60 projects do take some effort to manage (porting, updating, sending diffs upstream), were they not also worked on by many other developers (many of who I'll never meet), it would not be possible for me to provide nearly so much functionality to the system myself.

Clearly, then, open software offers a great value to businesses that have a use for it in their products.

### **Where Apple Adds Value**

Perhaps Apple could take NetBSD, clean up the Power Macintosh support, stick an Apple logo somewhere, and call this great thing "Mac OS X Server". But then, if you are looking for a good Unix server, you don't need Apple. You can instead quite readily install a NetBSD system on your Macintosh, or (gasp!) a PC, and carry on. We know, in fact, that this is satisfactory for many users. On the other hand, we also know that many of our users would not consider such a thing.

Apple's interest in BSD, therefore, cannot be as a replacement for Mac OS, but as a foundation for the rest of the system we provide: a polished, well-designed user experience. Apple can provide to its end-users a great deal more than current Unix systems offer. Our users expect a system that is easy to understand and use; one that is friendly; one with some style.

This goes well beyond user interfaces; it includes simple to set up hardware which "just works" with the included software, application tool kits for developers which allows them to provide a consistent user experience and give them the ability to leverage all of the services of the underlying system. It also includes five yummy colors.

Macintosh is about personality; it's about the people who use it, and the experience of using it. Combined with the advanced power and robustness of a new underlying foundation, that experience can be... uninterrupted, and all the more pleasant.

### **The Impact of Various Licenses**

There are many sources of open software and therefore many different licenses attached to open software. One of the compelling reasons behind using BSD as the Core OS is the simple BSD license. While an attached credit requirement takes a bit of work to honor, the license allows for any use of the software so long as the copyright is preserved. Companies looking for source with (almost) no strings attached would do well to use code under a BSD-style license.

A lot of companies would not bother to send their changes upstream, for fear of giving up some valuable information or technology, to send any useful code changes up to the upstream maintainers. Occasionally, these concerns have merit. More often, companies simply want a competitive edge, and aren't willing to share the wealth. One has to wonder whether this is good logic. The fact is that more diffs translates to more work. Eventually, the upstream provider will release a new version, perhaps with some important bug fixes and useful new features. If you have put significant work into improving the software, and kept it to yourself, you will have to merge these changes into the upstream source each time you want to update your code. This will cost you some developer time, perhaps enough that updating is not worthwhile, and your competitive edge can erode rather quickly. On the other hand, if you hadn't put much work in at all, there likely isn't much of an edge to lose.

I have made this argument successfully at Apple, and the result is that Apple started a very beneficial

technology relationship with NetBSD, in which we can easily update our code periodically, and NetBSD gets an occasional update from Apple as well. Mac OS X Server shipped with a developer CD including a fair amount of source code from NetBSD, (as well as the required source code for GNU software) so that our developers could help out as well with any problems they experience related to that code.

Another popular open software license is the GNU General Public License, which allows mostly unrestricted use of software so long as source code is available under the terms of the GPL for any changes made to the software. This requires a greater leap of faith for companies, and is often enough to discourage companies from using such software in commercial products. However, once having experienced the benefits of open software, the GPL is, in principle, an acceptable license. Unfortunately, the GPL's potential tendency to "infect" additional code keeps many companies (including Apple) wary of using GPL'ed software.

### **Being a Contributor to the Community**

As explained above, there is little reason for Apple to take open source software from BSD and treat it as proprietary software. Since the BSD subsystem is not the primary selling point of our complete system, nor a particularly unique subsystem given its free availability, Apple can only benefit by keeping our code in sync with the upstream source and thereby helping to improve the technology on which we depend. By contributing bug fixes and incremental improvements back to the open source community, we all win.

Apple's positive experience with open software in the recent past, and a belief on the part of Apple's executive team that the benefits of open software are indeed substantial, we have recently taken steps to contribute more substantial technology to the community by forming the Darwin project. It is our hope that by allowing our users and developers to work with us in improving our software, that Apple and its customers will see a great deal of benefit, and build a better relationship with each other. In addition, we hope that our efforts put us on the right track toward a healthy relationship with the greater open source community.

### **About the Presenter**

Wilfredo Sánchez is a 1995 graduate of the Massachusetts Institute of Technology, after which he co-founded an Internet publishing company, Agora Technology Group, in Cambridge, Massachusetts; he then worked on enabling electronic commerce and dynamic applications via the world wide web at Disney Online in North Hollywood, California. Fred is presently a Senior Software Engineer at Apple Computer in Cupertino, California. He works primarily on the BSD subsystem in Mac OS X Server as a member of the Core Operating Systems group, and as technical lead for the Darwin project. Fred is also a member of the NetBSD Foundation, Inc., and a contributor to the Apache HTTP Server Project.

### **Contact Information**

Wilfredo Sánchez  
Senior Software Engineer / Scientist  
Apple Computer, Inc.  
1 Infinite Loop / 302-4K  
Cupertino, CA 95014  
408.974-5174  
408.974-0362 FAX



# BUSINESS ISSUES IN FREE SOFTWARE LICENSING

Donald K. Rosenberg, *Stromian Technologies*

## I. BACKGROUND: FREE SOFTWARE MOVEMENT AND SOFTWARE VENDORS

There are some odd ideas circulating about Linux and the Free Software or Open Source movements. We can call these ideas primitive because they are simplistic and not well thought-out, and because they go back to the *reductio ad absurdum* of the primitive peoples who believed (and still believe, we hear) in what anthropologists like to talk about as the “cargo cults.”

According to the anthropologists, these movements began among South Seas peoples in the 19<sup>th</sup> century, when they awaited the arrival of large ships which would restore to them all the wonderful goods their peoples had owned once, long ago. After World War II these cults took the form of waiting for aircraft to descend from the skies with their abundant cargoes. We are told that the believers even constructed runways with mock aircraft on them, hoping to attract the passing air traffic.

We all smile—how much more we know than they—but today there is a firm body of thought on the one hand that eventually all software in the future will be produced, shared, and enjoyed on the Bazaar model: freely developed and given away by loosely-organized programmers around the world, and superior in quality and design to the commercial products of today. Given the behavior of much modern commercial software, one can understand why the believers hope so fervently for the millennium.

But commercial vendors are just as likely to make the same mistake: we see articles and columnists hyping the idea that if a software firm can just take the leap of faith into arms of Open Source, they will attract legions of the world’s smartest programmers, working ceaselessly and without compensation to improve the code the vendor has thrown among them. It does not help that any announcement that a company is releasing source code is regarded by the business community as a desperate act of last resort.

What should be the approach of a commercial software vendor to the Open Source space? And what do they really want, anyway?

## II. WHAT ARE SOFTWARE VENDORS TRYING TO ACCOMPLISH?

Software vendors want to a) protect their financial investment in their code, product, and channels of distribution and b) to recover that investment, along with a profit. Somehow they want to make sure their work is not appropriated, and that there will be revenues which will keep the doors open and the families fed.

These two themes—protection of property and recovery of investment—will dominate the rest of the talk.

## III. HOW CAN THEY DO IT?

### A. Current Open Source Models

Protecting the property and tapping the correct place in the distribution stream for revenue are the chief purposes of commercial software licensing. There is a great variety of licenses available—the key thing to remember is that they should reflect the business goals of the vendor.

#### *GNU General Public License*

The goals of the Free Software Foundation are to keep its GNU software (and any other software using the GNU GPL) completely free and Open Source. Nevertheless, many vendors can make money from it by providing related services. Two highly-successful examples are O’Reilly & Associates, who publish books about Open Source products, and Red Hat Software, a branding company described by its Chairman, Bob Young, as “a company that gives away its software and sells its sales promotion items.” Red Hat distributes the Linux OS for money on CD (as well as for free from its Web site), and like numerous other companies, sells software support.

The GPL is a good thing for an operating system: it is something that all users want kept free and open; all developers want equal access to the system so that none can take advantage of operating system secrets. Most tool and application vendors, however, will not want to go into the Open Source arena until they understand how it works, and how they are going to structure their company, products, and services to make money in this arena.

There are different ways to go about this; to pick just a couple of examples of products that make their source code available for free, let’s look at Scriptics and Aladdin. Both succeed by segmenting their product

line into free product and revenue product, on the basis of licensing.

### *Scriptics*

The popular scripting language and toolkit, Tcl/Tk, claims a million users. Scriptics is a new firm founded by the developer to commercialize Tcl/Tk. While keeping the core material and its improvements free, the firm will develop niche applications for money. Because the user has the right to modify and distribute the source code, John Osterhout recognizes that he has to keep the free side of his business happy in order for Scriptics to remain the center of Tcl/Tk development. He intends to do this by keeping the scripting language and toolkit good enough so that the free side attracts new users. In turn the Scriptics Web site aims to be the principal online resource for Tcl/Tk, attracting prospects and converting them to users. Profits from the commercial side of the operation will pay for the in-house developers working on Open Source Tcl/Tk.

Thus Scriptics divides the free/revenue sides of the business by focussing one on core technology, and the other on niche adaptations of or extensions to that core.

### *Aladdin*

Aladdin's Ghostscript splits itself into different free/revenue versions using different licenses: a separate enterprise called Artifex distributes a commercial version called Aladdin Ghostscript, and there are free versions under the Aladdin Free Public License and GNU Ghostscript. The Aladdin Free Public License resembles the GPL and has additional restrictions. You can't accept money for the free program except for cost of disks and copying, you can't put the free version on a disk with any paid-for software; the bundling restriction helps kill commercial distribution of the free product. On the other hand, the licensed commercial user can use Ghostscript in his application and also get interim updates to the code; free users wait for the annual update.

Thus Aladdin divides the business into free/revenue by segmenting the technology into core technology vs. paid latest updates to that core.

But these examples do not mean that any vendor can release the source code for a product, and expect the community to enthusiastically pick it up and improve it. The vendor needs to provide a good infrastructure for maintaining the source tree, including bug lists and version control, and in all the enterprises named above—Red Hat, Scriptics, and Aladdin—there are paid staffs of programmers who maintain and improve

the source code, and then make it available (at some point) for free to the community. All of this effort is not merely a matter of being seen to cooperate with the Open Source community by making contributions; it is all part of maintaining a leadership position as the authoritative source of the free product.

Different licenses are used to maintain different degrees of control in upholding this leadership. Red Hat and Scriptics ride bareback: Red Hat faces competition from other Linux distributions, and by offering a better version of Tcl/Tk, it is theoretically possible for another party to make Scriptics a secondary player in the free version of Tcl/Tk. Much more restrictive are the recent licenses such as the Sun Community Source License, which makes it clear that although the source is open to examination, modifications will be tightly controlled by Sun through a testing and revenue-license program, and that only Sun may maintain a source tree for the code. It will be interesting to see what success these restrictive licenses will enjoy.

Restrictive as the Sun Community Source License is, it is at least fairly clear when it comes to the user. A more difficult problem has been introduced into the Linux community by the free or public versions of the license for the Troll Tech Qt library.

### *Q Public License*

The library is a toolkit: the user is free to use and distribute it and his derivative application--unless it's a commercial application. Commercial distribution is effectively stopped by requiring the derivative application to give away its source code and permit further distribution and modification. Commercial users must buy the Qt toolkit under the Professional Edition License; the developer pays a round sum for the Qt toolkit and the right to distribute runtimes.

The chief objection to the old Qt Free Edition License was that it did not permit the Qt toolkit itself to be modified in any way when it was distributed. The license also contained incorrect and confusing language about use of the GPL or a BSD-type license as alternatives. These are the perils of writing your own license and not getting it right. The new QPL, however, has cleared away the confusion and now also permits the distribution of separate patches to the Qt source code, so the Open Source Initiative (OSI) ([www.opensource.org](http://www.opensource.org)) says the license now meets the Open Source Definition (OSD).

There are still plenty of voices objecting, however, to Qt's licensing, even under the new QPL, and the voices are raised because Qt is the technology underlying

KDE, the highly successful free Linux desktop. Note that a developer can never use the Q Public License on anything but some sort of UNIX platform (basically Linux), and that Windows (and the Mac) are reserved for the Professional Edition.

The motives are clear: Troll Tech wants to control the toolkit, and to prevent forking; therefore, no modifications are permitted. The firm makes the product free on Linux in hope of collecting improvements from users, and wants to reserve the Windows and Macintosh platforms for their revenue product. Troll Tech wants to use the Q Public License to promote their technology, spread its use and familiarity, and lure people (some will say trick them) to the Professional Edition.

## **B. Licensing Dependencies**

In some respects the business model for the Qt toolkit is not unlike other free/commercial segmentation. If you want to distribute for money, you pay Troll Tech money. But the stinger is that persons developing software on the free and highly popular KDE desktop suddenly find that they owe money to a third party, Troll Tech, for use of the underlying QT toolkit. As long as the application was free, there was no problem; as soon as the developer wanted make money (or even just to collect a little revenue to cover his costs), the licensing terms undergo a change of state that is working to undermine the formerly unified Linux desktop.

This is an issue of license dependencies; it can be a problem as bad as software dependencies.

In the case of Qt, the solution to the dilemma may not include the survival of Troll Tech. Although some people think Qt is such great technology that Troll Tech will be able to get what they want, others are so horrified by this problem that we have forking in what had seemed to be a common Linux desktop. Debian dropped Qt and KDE from its distribution, posting an explanation on the site; Eric Troan explained on the Red Hat site that Red Hat could not put Qt and KDE software on the basic development CD if its licensing terms were so different from the other development software there.

Finally, resentment about Qt's licensing has caused movements to spring up to clone Qt. Harmony, a project to clone a Free Qt, is still active, I've been told, and the GNOME movement has sprung up to put out a rival toolkit distributed under the GPL and LGPL, just like Linux. There are desktop projects based on GNOME, and Red Hat is working on one of them. It

will take the Linux community a while to get over this split, which has its origins in licensing dependency.

At the end of this paper is a diagram of a simple scheme of purposes and dependencies:

### **Base Layer – Operating System – GPL**

Operating systems stand to benefit the most from the GPL because they are the broadest-base software; the users of an operating system will always outnumber the users of any particular application on that system. There is more choice in applications than in operating systems. The GPL may do its best work at this level, forcing a standardization of all licenses, and aggressively keeping it open and free of all closed material. Openness is highly important for operating systems; applications can get away with more-limiting licenses because their usage is more limited.

In this Base Layer of the OS, vendors can earn money on source code distribution, and they can earn money on binary distribution (so long as source code goes out with it). Linux distributions using the GPL are more or less successful businesses

### **Second Layer, Part A: Toolkits**

At the next level, software may be either free or commercial, but it is essential that there be a firewall here (the vertical red division); the same toolkit should not be capable of changing state. The firewall represents not a separation of products for free/commercial, but clear licenses for those products. Tcl/Tk, for instance, lets you use it for commercial products or for free products.

### **Second Layer, Part B: Extensions and Libraries for Toolkits**

At this next higher level, extensions and libraries need to adopt the same licensing (either free or commercial) as the toolkits they serve. Otherwise we're back in the "checkerboard" or "change-of-state" license. Developers must watch licensing dependency as closely as software dependencies.

### **Third Layer: Tools and Applications**

Of the applications which are not tools, we can expect a larger proportion of these to be binary or proprietary. The proprietary applications can build upon both

proprietary and free foundations, provided they respect the licensing of the layers upon which they are built.

Tools, however, need to follow the same choice as toolkits, being either on the Free side or the Proprietary side, so that their products likewise have unambiguous licenses.

At this third level, vendors have a strong desire for proprietary code to protect their development investment, and distribution in binaries is common for many products. The LGPL is often not enough to allow for use of proprietary code, and so BSD-style licenses fit here, as do products like the Apache-based Stronghold. The Aladdin Free Public License operates at this level in the Free category, while the Perl Artistic License, which is useable for closed, embedded commercial work fits into the Free and Proprietary category (we might ask whether this license would work as well for an application as it does for a language/script).

### **C. Further Innovation and Cooperation Between Commercial and Open Source Software**

So far we've talked about some of the common ways a software vendor can ally with the Open Source movement (such as distribution and support), and we've seen that a license must be carefully written to achieve the business goals of the licensor. We've also seen that it is not a matter of tossing the software out there and expecting brownies to work on it during the night—a software vendor must provide the infrastructure and encouragement in order to form a coding community around the software.

This brings us to the question of a strategy for use of Open Source to gain advantage for proprietary software. In a world in which a technology must be brought to market quickly and just as quickly achieve nearly universal distribution merely in order to survive, there are many products which are stymied by the chicken-and-egg problem: how to get the user or client piece out there, when there are few users of the server component, and, reciprocally, how to get the larger server engine adopted, when it cannot be demonstrated that there are many correspondent client pieces out there waiting to be served. The traditional approach to this is to give the client piece away, hoping that low price will make rapid adoption more likely. The vendor still has the obligation to maintain and distribute the client piece.

I would like to suggest that an Open Source model could be adopted for the client piece. To take a

concrete example, the VRML standard for 3D viewing over the Web (currently being renamed and updated into the Web3D standard), depends on users having a VRML browser attached to their Web browser. For a variety of reasons, efforts at developing and distributing VRML-standard browsers have not been successful, and have been largely abandoned. A company wishing to promote a server-side (that is, development) VRML product should release an Open Source VRML browser project. Because the browser would be useful as a reader for all VRML objects, not just those from the particular vendor, the developer community might well take an interest in perfecting and maintaining such a browser, because it would have uses far wider than serving a single vendor. The vendor's benefit would be seeing the VRML market expand, and growing close ties to the VRML developer community that would both provide Open Source support for the browser, and be the likeliest customer pool for the server product.

There is room for all sorts of combinations of profitable cooperation between commercial software vendors and Open Source software—we have hardly begun to try them all.

## Layers/Licenses

**Operating System === GPL**

**Toolkits;  
Extensions**

[Free]

**Toolkits;  
Extensions**

[Proprietary]

**Tools**

[Free]

**Applications**

[Free and Proprietary]

**Applications**

[Proprietary]

**Tools**

[Proprietary]



# Sendmail Evolution: 8.10 and Beyond

Gregory Neil Shapiro  
gshapiro@sendmail.org

Eric Allman  
eric@sendmail.com

*Sendmail, Inc.*  
6603 Shellmound Street  
Emeryville, California 94608

## ABSTRACT

*Sendmail*<sup>TM</sup> has been the *de facto* mail transfer agent implementation since the dawn of the Internet. Today, *sendmail* development is still driven by a continually changing set of network requirements and user demands. Lately, two new driving forces have also contributed to *sendmail* development. First, as more open source mail transfer agents, such as *Exim* and *Postfix*, become available, a new friendly competition has developed in which the authors of the various MTAs share their ideas via open source and help to advance open standards as opposed to advancing their own particular implementation. Second, a new “hybrid” company, Sendmail, Inc., has been created to offer commercial versions of the open source software while continuing to fuel open source development.

This paper will briefly discuss the evolution of *sendmail*; the influences which drive *sendmail* development; and how the creation of Sendmail, Inc. has contributed to the open source version. The paper will also describe the new features appearing in the next “functionality release” of open source *sendmail*. In particular, changes in queueing and new protocol support are discussed. Finally, the authors will speculate on future directions for *sendmail*.

## 1. Introduction

The *sendmail* mail transfer agent (MTA) is used on most UNIX<sup>TM</sup> systems today. Recent changes have influenced *sendmail* development, notably the creation of a new “hybrid” company dedicated to supporting both the open source code as well as a commercial version.

Section 2 gives a brief history of *sendmail*. Section 3 describes the forces acting to influence changes in *sendmail*. Section 4 outlines Sendmail, Inc.’s effects on the open source. Section 5 discusses changes appearing in *sendmail* 8.10. Future directions that *sendmail* may take are laid out in section 6. Finally, a summary and concluding remarks are presented in section 7.

## 2. History

To understand the continuing evolution of *sendmail*, you must first look at its history. Like many successful open source projects, *sendmail* started as a “scratch your itch” solution to a problem.

### 2.1. In the Beginning...

*Sendmail* started out as *delivermail*, written by Eric Allman, then a graduate student and staff member at the University of California at Berkeley. *Delivermail* solved the problem of routing mail between three different networks running on the Berkeley campus at the time: the ARPAnet, UUCP, and BerkNet. The first public version was distributed in 1979 as part of the Fourth Berkeley Software Distribution (4BSD) and later as part of 4.1BSD [Allm85].

Although *delivermail* solved the immediate problem faced by Berkeley, it was not generic enough to solve the problems of other custom networks in operation. Since the instructions for talking among the networks were part of the C source code, it was not easy for sites to reconfigure *delivermail* for their specific needs. The configuration was also not flexible enough to handle complex mail environments.

At the same time the ARPAnet was transitioning to the new Internet protocol, TCP/IP. Part of the new protocol suite included extracting mail transmission out of the file transfer protocol (FTP) into its own protocol, the Simple Mail Transport Protocol (SMTP) [RFC821].

The user demand for a customizable program and the network requirements created by the new mail protocol led to the creation of *sendmail*, which first shipped in 1983 with 4.1c BSD—one of the initial operating systems to support TCP/IP. *Sendmail* accomplished two important goals. First, it provided a reference implementation of the Network Working Group (later the Internet Engineering Task Force, or IETF) mail standard [Cost97]. Second, the configuration was read at run time to allow reconfiguration for different networks without recompilation. Because of the wide variety of networks supported, the configuration was designed to be friendly toward non-conforming addresses. Instead of rejecting messages that were not acceptable to the standard, it tried to repair them; this broad acceptance of inputs maximized interoperability with other networks available at the time, such as UUCP.

By late 1986, Allman's involvement with *sendmail* had tapered off, and several other people picked up development. The most important version was *IDA sendmail* from Lennart Löfstrand of the University of Linköping in Sweden, with later maintenance by Neil Rickert of Northern Illinois University and Paul Pomes of the University of Illinois [Cost97]. The most important feature added by IDA was the concept of external databases in DBM format. Shortly thereafter, Paul Vixie, then at Digital Equipment, created *KJS (King James Sendmail)*, an attempt to unify the divergent versions, but this version was not widely adopted. *Sendmail* had effectively splintered.

## 2.2. Sendmail 8 Emerges

In late 1989, Allman returned to U.C. Berkeley, and not long thereafter was drawn back into *sendmail* development. By July of 1991, serious work on what would become *sendmail 8* had begun. Many ideas were taken from *IDA sendmail* and *KJS*, although most were generalized. For example, external databases were

added, but in such a way that formats other than DBM were available. *Sendmail 8.1* was released with 4.4 BSD in mid-1993. *Sendmail 8* quickly became a unifying influence, as vendors converted from their hacked versions to the newer version. Some features from vendor versions were also included in the new release, for example, NIS support from Sun Microsystems. These additions are just one of many examples of the success of open source software: *sendmail 8* was fertilized with ideas from other open source and vendor versions.

Another important change that occurred concurrent with *sendmail 8* was that versions were controlled more carefully. The previous major release (*sendmail 5*) had no fewer than 143 “dot” releases (that is, 5.1 through 5.143), often more than one in a single day. Some of those were intended for public consumption, some were test releases. With version 8, *sendmail* switched to a policy of clearly labeling test releases, producing production releases less often, and clearly identifying new functionality releases from bug-fix releases. This change in release frequency was essential to the wide acceptance of *sendmail 8* by the community. The downside of this change is that people who like to be on the “bleeding edge” have to wait longer, and new features are not tested immediately. We view this loss of quick feedback as being an acceptable tradeoff.

An unfortunate effect of the success of *sendmail 8* was that Allman quickly became overloaded with answering questions. This overload was the impetus behind the establishment of the Sendmail Consortium, a loosely-knit group of volunteers providing free support for *sendmail*. Gregory Shapiro was invited to join that group during the 8.8 cycle, and by 8.8.6 was doing a large part of development and most of the release engineering, although Allman continued to review and approve changes.

In 1997, Allman found that even with the help of an extremely capable volunteer staff, he was unable to keep up with the support load and continue to move *sendmail* forward. After exploring several other approaches for adding resources for *sendmail* development, he finally settled on founding a “hybrid” business model company to produce a commercial version of *sendmail* while continuing to support and extend the open source version. By using the “hybrid” approach, he was able to protect the interests of the open source community while creating a viable business model.

### 3. Driving Forces

As can be seen in the preceding section, *sendmail* has responded to both changing network requirements and user demands. In addition to these demands, new open source MTA alternatives help in driving *sendmail* forward.

#### 3.1. Network Requirements

The network requirements come both from the changing face of the Internet and from new Internet drafts and RFCs from the IETF. For example, up until version 8.9, *sendmail* allowed third party, promiscuous relaying by default. This willingness to relay had been an acceptable, even desirable, default for more than 15 years. Unfortunately, with the growth of spam on the Internet, this default is no longer acceptable.

The increasing use of email as a vector of viruses has heightened the need for MTAs to include content checking. An SMTP server running on a firewall must be prepared to vet the data it is handling. Because of this need, 8.9 included message header checking and 8.10 will include a mail filter API for more advanced header and body filtering.

Changes in Internet standards from the IETF also have a major impact on *sendmail*. During 1998, the IETF accepted 22 new RFCs that involved electronic mail in one form or another. At least one of these RFCs has a direct effect on MTAs such as *sendmail*. RFC 2476, *Message Submission*, specifies a separate protocol for initial insertion of a new message into the message delivery system using SMTP [RFC2476].

As of April 1999, three more MTA-related RFCs have already appeared. RFCs 2487 [RFC2487] and 2554 [RFC2554] provide encryption and authentication for an SMTP session. RFC 2505 [RFC2505] is a set of recommendations for features that MTAs should provide to combat spam. Additionally, the Detailed Revision/Update of Message Standards (drums) IETF Working Group is preparing to release the long-awaited update to RFCs 821 and 822, probably later in 1999 [SMTPUPD, MSGFMT]. Clearly, the messaging standards landscape is not static.

#### 3.2. User Demands

By far, however, most feature requests come from *sendmail* users. It is common for the Sendmail Consortium to receive three to five feature requests per week, some complete with the patches necessary to implement the feature. These feature requests produced a list of 320 requests before 8.10 development even began.

When deciding which features to implement and how they should be implemented, we try to balance backwards compatibility with change. By introducing radical changes gradually, we give *sendmail* sites a chance to prepare for the changes. A combination of a huge user population and 20 years of *sendmail* availability prevents us from doing radical changes without advanced warning. For example, the 8.9 documentation included a notice warning users that configuration file names would be changing in 8.10. Also in 8.10, the LDAP map class will be changed from `ldapx` to `ldap`, thereby dropping the class name's connotation as an experimental map. The old name will continue to work (and print a warning) in 8.10, but will be removed in a subsequent release. Some of the other open source mail transfer agents, such as *Postfix* and *qmail*, are not yet so constrained.

#### 3.3. Alternatives to Sendmail

At the same time, these other open source MTAs also drive *sendmail* development. The open source alternatives, such as *Postfix* and *Exim*, give *sendmail* a (for the most part) friendly form of competition. This competition promotes both innovation and sharing for all of the MTAs. For example, Wietse Venema, author of *Postfix*, not only asked about *sendmail* behavior during his development of *Postfix* (to maintain compatibility), but also made contributions to *sendmail*.

With this mostly friendly competition and cooperation among open source authors, everyone wins. Without multiple open source implementations, there would be no choice for the user, nor much pressure to move the existing implementation forward or adhere to standards issues. With multiple implementations, users are free to choose the open source MTA with which they are most comfortable. Since the MTAs are all based on open standards instead of commercial, proprietary standards, they are able to interoperate and prevent the Internet from becoming proprietary and vendor specific.

### 4. Enter Sendmail, Inc.

As one might imagine, the creation of Sendmail, Inc. represents a major change in the development of the open source version of *sendmail*. Now, there is a commercial entity behind the development—a company that is completely committed to the open source. The development of *sendmail* would have continued without the creation of Sendmail, Inc., but at

a slower pace and with fewer resources. Sendmail, Inc. was able to release its first commercial product, Sendmail Pro<sup>TM</sup> quickly and successfully thanks to the already available and proven open source *sendmail*. In return, Sendmail, Inc. can contribute both financial and human resources to open source development. These contributions can be found in the many places within the company.

#### 4.1. Engineering

As detailed in the history section, up until the formation of Sendmail, Inc., all *sendmail* development and support was done by volunteers in their spare time. This development model limited the total energy that could be exerted. Sendmail, Inc. has been able to create a complete engineering team to work on *sendmail*, including software engineers, quality assurance (QA) engineers, and technical writers.

There are currently two full-time engineers working on the next version of the open source, Gregory Shapiro and Claus Assmann, with more to be added as they can be hired. These new engineers will help deal with the growing complexity of new standards and respond to new user requests as they arrive. Additionally, other engineers are working on commercial products, and selected features from those products are being included in the open source distribution.

The presence of a QA department has an additional impact. Previously, formal testing was minimal; in particular, formal testing tools, such as code coverage tools, were not applied by *sendmail* core developers. The Sendmail, Inc. QA department now provides the first line of formal testing before release to outside testers.

The technical writers provide professional writing and editing resources to improve and expand the available documentation. They will be able to help clean up and augment the available documentation for the open source *sendmail* distribution.

Beyond people, Sendmail, Inc. has made commercial development and software testing tools (such as memory leak detectors and code coverage monitors) available to the engineers. These tools were previously too expensive for the volunteer developers<sup>1</sup>. The company is also able to afford a variety of

---

<sup>1</sup>It would seem a "good thing" if producers of software development tools would consider donating copies of their software to established open source development groups that could not otherwise afford them.

hardware platforms giving development and QA engineers the chance to test portability in-house before releasing a distribution for testing.

This "hybrid" approach to *sendmail* development introduced some new concepts for the developers, for example, project schedules and structured code reviews. Previously, *sendmail* development was not done according to a schedule and there were no hard deadlines<sup>2</sup>. Releases were made when they were ready instead of on a predetermined date. The addition of commercial influences does not mean that releases will be made before they are ready. Instead, given a future release date, the number of features that can be implemented in that time frame are determined. Of course, if there are problems with the release as the release date nears, either those problematic features will be removed before the release or the schedule will slip. No version will be released before it is ready.

As more engineers work on the code, more structured code reviews are planned. Previously code review was done on an ad hoc basis; for example, Allman reviewed most newly contributed code and someone (often Shapiro) reviewed most of Allman's code. Now all code check-ins are mailed to a list of core team members. This mailing has the effect of keeping everyone "in sync" and catching many problems immediately. For wholly new code, more formal code reviews will be instituted.

#### 4.2. Support

The support infrastructure of Sendmail, Inc. collects and reports problems, analyzes trends of incoming questions, and provides feedback to the developers. Sendmail, Inc. consultants and engineers now visit customer sites, allowing them to see *sendmail* in use in the field and discuss the customer's needs and expectations. These visits will all lead to improved features and clearer documentation not only for the commercial customers, but for the open source users as well.

Another unexpected benefit has been acceptance by companies that refuse to use "free software" because of fears that they will not be able to get support. Particularly for "mission critical" code such as the mail system, many companies require a commercial support organization that has contractual obligations to answer questions within a certain time frame, and as a last resort, an entity willing and able to stake its reputation

---

<sup>2</sup>There were targets, for example, *sendmail* releases were often targeted to precede USENIX conferences.

on the ability to provide solutions to customers' problems. Sendmail, Inc. can provide guaranteed 24/7 support coverage.

### 4.3. Other Expertise

Sendmail, Inc. also provides specialized resources for handling other tasks unrelated to the software itself, freeing up the developers to do what they do best. Although *sendmail* has always had some form of "marketing" to entice users to upgrade, it has not had a marketing organization to spread the gospel and inform trade press about the features of new versions. That has all changed with the creation of Sendmail, Inc. and a true marketing department. For example, with the formation of the new company and the public beta release of *sendmail* 8.9 in March of 1998, information about the release and its new anti-spam features made the front page of the *New York Times* [Mark98].

With the addition of a business development unit, Sendmail, Inc. is in a better position to partner with other companies to provide enhanced services to the *sendmail* community. For example, third-party commercial virus and spam checkers are planned for availability with *sendmail* 8.10. Since the hooks needed for these third-party plug-ins will be in the open source release as well as the commercial release, open source users will be able to take advantage of these new filtering technologies.

Finally, Sendmail, Inc. provides the legal resources necessary to research and complete the necessary paperwork for the open source distribution, such as licenses and government export approval for features like SMTP authentication and secure SMTP (discussed below).

## 5. The Present: Sendmail 8.10

Even though coding for *sendmail* 8.10 began the first week of February, 1999, plans for the version started even before 8.9 released. The 8.9 release was to be the anti-spam release and there was great demand to get these features out to the users as soon as possible. This time pressure forced us to limit new functionality to spam fighting features and defer others for the next release. We also did not want to obsolete the "Bat Book" [Cost97]; which would be a disservice to our users and the open source customers. From the 320 customer requests, we picked more than 100 features for inclusion in 8.10; another 80 were selected as

potential features if time permits in the release cycle.

As with past releases, 8.10 has a "theme". Although many of the other changes are important, we plan to highlight SMTP authentication and a new mail filter API as the premier features for 8.10.

### 5.1. SMTP Authentication

Our hopes are to have SMTP authentication [RFC2554] as part of the 8.10 release. SMTP authentication provides a method for the mail user agent (MUA) to authenticate the user to the mail transfer agent and carry that authentication with the message as it passes between mail servers toward the final destination.

For message submission from an MUA into a site's mail server, SMTP authentication provides a mechanism for recognizing users as trusted for that site. This feature can be used to allow relaying based on the submitting user instead of the submitting host, a feature especially useful for roaming users submitting mail from untrusted sites.

Although the authentication information is carried in the message envelope until reaching the final delivery host, remote sites should not trust this information as it may have been altered by a "man-in-the-middle" attack. As the RFC notes, SMTP authentication is "generally useful only within a trusted enclave" [RFC2554]; it is not meant as an end-to-end authentication or security mechanism.

Initially, *sendmail* will use the message submission authentication to override the relaying checks. It will also provide the authentication information to user rulesets as macros.

Unfortunately, there is potentially a major road block that would prevent us from including SMTP authentication—the United States government's cryptography export policy. Although authentication is claimed to be acceptable for exporting, the Bureau of Export Administration may reject our application if the bureau feels the authentication hooks in *sendmail* can be easily converted to provide encryption, even though enabling encryption is not the purpose of the hooks. The definition of "easily converted" is unclear. Surprisingly, the distribution of *sendmail* in source code form hurts our chances of getting approval. Products that do not ship with source code, such as the Netscape<sup>TM</sup> Messaging Server, are able to ship SMTP authentication. In such cases, those products are able to limit the use of the routines to an authentication model that is weak enough to be accepted by the United States government. Also, in binary form, it would be nearly impossible to convert the authentication routines

into encryption routines.

Assuming we are able to distribute 8.10 with SMTP authentication, there are still some outstanding issues. In creating the extension, we needed an implementation of the Simple Authentication and Security Layer (SASL) [RFC2222] library to provide the framework for the different authentication methods. The only open source implementation currently available (that we are aware of) is the Cyrus SASL library. Early attempts using this library were not encouraging because of portability and implementation problems. Since we do not currently have the time or resources to create our own SASL implementation, we have decided to use the Cyrus library and contribute all our bug fixes and portability changes back to the Cyrus development group. These changes are being incorporated into the base Cyrus release, making them available to others and providing a stronger implementation for use with *sendmail* when 8.10 is released. This feedback of changes is another example of Sendmail, Inc.'s commitment to the open source community.

## 5.2. Third Party Mail Filter API

The other major functionality enhancement for 8.10 is a third party mail filter API. This API will allow system administrators and third party companies to provide message filtering using hooks in the *sendmail* code. This new plug-in architecture will allow for better spam and virus monitoring as well as give administrators the ability to accept, reject, discard, modify, or archive messages.

Briefly, *sendmail* will have a compile flag that will implement callouts to user-supplied routines that will be called to process envelope information, headers, and the message body. The filter can request that new headers be added or the entire message body be replaced. In addition, portions of the envelope can be modified—in particular, recipients can be added or deleted. This API provides exceptional flexibility.

## 5.3. Other Noteworthy Changes

Beyond these two new major additions, 8.10 will include many other new features. Although it is impossible to describe them all in this paper, we will mention some of the high points.

**IPv6 support.** In response to network changes, *sendmail 8.10* includes IPv6 support using the interface described in RFC 2553, *Basic Socket Interface Extensions for IPv6* [RFC2553]. This support allows sites that are moving to IPv6 the ability to include *sendmail* in their transition plans and testing.

**New RFC support.** Other RFCs under consideration for possible *sendmail 8.10* inclusion are the message submission protocol [RFC2476], enhanced SMTP status codes to provide more precise error reporting [RFC2034], and anti-spam recommendations [RFC2505].

*Sendmail 8.9* is already compliant with RFC 2505, *Anti-Spam Recommendations for SMTP MTAs*. However, we are investigating including a suggested change so that the MTA can limit the maximum number of messages that can be sent from a particular user in a specified interval. This feature will help to reduce the damage that can be performed by “hit-and-run” spammers.

**Improved virtual hosting capabilities.** The most requested enhancements has been for better virtual hosting support. *Sendmail 8.10* will include better control over the virtual user table, which provides a domain-specific form of aliasing, allowing multiple virtual domains to be hosted on one machine. A new class is available for triggering virtual user table lookups to match the functionality of the generics table, the feature used to rewrite local addresses into a generic form. “Plus detail” information, the portion of the mail address used to carry additional information about the user address that precedes the plus sign (for example, *user+detail@host*), will also be made available for both generics and virtual user table lookups.

Additionally, 8.10 maintains information regarding the incoming connection in a new macro. For example, hosts having multiple IP addresses on different virtual interfaces always advertise themselves as the primary host name in 8.9. In 8.10, they will be able to identify themselves as the virtual host throughout the transaction. The SMTP greeting and Received: headers will use the virtual host name and outgoing IP connections will be bound to the address of the customer instead of the hosting ISP (so the “next hop” SMTP server will log the appropriate host name in its Received: lines).

For sites providing queueing services, 8.10 will offer a new mailer flag for queueing mail until delivery is explicitly requested via either a queue run with pattern matching (*-qR*, *-qS*, *-qI*) or via ETRN, the SMTP service extension for remote message queue starting [RFC1985]. This feature provides better support for ISPs that provide queueing for dial-up customers, as queue runs are no longer held up waiting for the dial-up server connection attempt to time out.

**Improved anti-spam features.** To allow users more fine grain control, 8.10 introduces more detailed specification for the access database. Tags on the key

of access database entries can limit the lookups to specific anti-spam checks. For example, specifying `To:friend.example.com` instead of `friend.example.com` in the access database, allows relaying *to* friend.example.com without permitting mail relaying *from* that site.

A new DNS-based blacklist feature (`dnsbl`) supersedes the Realtime Blackhole List (`rbl`) feature available with 8.9. The new feature takes the name of the blacklist server as well as an optional rejection message. The blacklist server is queried with the IP address of each incoming connection and, if the query is successful and the IP address is blacklisted, the connection is rejected. This new feature can be included multiple times to allow sites to subscribe to multiple servers. `FEATURE('dnsbl')` replaces `FEATURE('rbl')` to prevent the possible confusion between the Realtime Blackhole List and other DNS based blacklist servers.

Other new anti-spam features include `FEATURE('require_fqdn')`, which requires a fully qualified domain name for sender addresses unless the connection comes from a local system, and `FEATURE('relay_mail_from')`, which allows relaying if the mail sender is listed as RELAY in the access map.

The ability to delay anti-spam checks until the SMTP RCPT command has been added using `FEATURE('delay_checks')`. This feature allows sites to permit mail to certain addresses, such as postmaster, regardless of the results of other anti-spam checks.

**New macros, rulesets, and options.** The 8.10 release also introduces new named macros and rulesets for controlling other facets of the daemon. Examples of the new macros include `${rcpt_mailer}`, `${rcpt_host}`, and `${rcpt_addr}`, which represent the resolved triplet that delivers the mail to this recipient. These macros can be used to simplify matching in custom `check_*` rulesets. Three new ruleset calls, `check_etrn`, `check_expn`, and `check_vrfy` have been added to restrict the ETRN, EXPN, and VRFY SMTP services. Instead of globally turning these services off via the `PrivacyOptions` option, administrators can now use the rulesets to allow these commands for certain sites.

New options have been added for general mail server policy and protection. These options include the popular `MaxHeadersLength` and `MaxMimeHeaderLength` options, which protect against “denial-of-service” attacks and buffer overflows in some MUAs.

**Better LDAP integration.** The 8.10 release offers tighter integration with the Lightweight Directory Access Protocol (LDAP), which has proven to be the directory service of choice at many sites. Support for multiple entry/attribute LDAP value searches, LDAP authentication, and LDAP-based alias maps will appear in *sendmail 8.10*. We are also monitoring the IETF LDAP Schema for E-mail Routing “birds-of-a-feather” group for a standard schema for alias specification using LDAP [LASER].

**Improved performance.** In an effort to improve *sendmail*’s performance, 8.10 includes code donated by Exactis.com (formerly InfoBeat<sup>TM</sup>) that provides support for multiple queues in *sendmail*. The new Exactis.com donation also includes code that extends the queue file name, making it unique for a 32-year period. This change reduces the amount of file locking and renaming necessary for instantiating a queued message. In addition, the new queue file naming system makes it possible to move items between different queues easily and quickly.

Exactis.com also donated the code necessary to implement memory-buffered files on systems that include the Chris Torek `stdio` library, such as the BSD family. If your operating system can take advantage of this new code, *sendmail* will be able to reduce file system overhead by not creating temporary files on disk. In combination with the new queue file naming system and multiple queue support, file system bottlenecks will be greatly reduced.

**Features from Sendmail Pro.** Beyond funding for development from Sendmail, Inc., the open source version also benefited by receiving new MTA features from Sendmail Pro, the commercial product. These changes were released in open source even before Sendmail Pro was released<sup>3</sup>. Two of these changes are new daemon control functionality and trusted user support.

The first, new daemon control functionality, allows an external program to control and query status from the running *sendmail* daemon via a named socket, similar to the `ctlinnd` interface of the INN news server [Salz92]. Although only a few commands (`restart`, `shutdown`, and `status`) are available in this first version, the framework is in place for extending this functionality to control and query different facets of the daemon. Since access to this interface is controlled by the UNIX file permissions on

---

<sup>3</sup>They were disabled by default as they had not been fully tested at the time of the open source release.

the named socket, the file permissions provide administrators a means of controlling the daemon via external interfaces without requiring root privileges. A Perl program (`contrib/smcontrol.pl`) is provided in the distribution as example code to take advantage of the control socket.

The new `TrustedUser` option allows the administrator to specify a user name that will be considered equivalent to the superuser for permission checks and other operations normally reserved only for root. For example, the `TrustedUser` is allowed to start the daemon as well as own maps, files, and directories without `sendmail` marking them as untrusted. This change is another step in the migration toward a `sendmail` daemon that does not heavily rely on superuser privileges.

**Consistent file names.** As of 8.10, the default location for all `sendmail` configuration files will be `/etc/mail/`. This change avoids sprinkling potentially dozens of files in `/etc` with obscure file names, such as `sendmail.cw` (now known as `/etc/mail/local-host-names`), and allows that directory to be owned and managed by the user specified in the `TrustedUser` option. The files affected include maps, aliases, and classes, as well as the error header, help, service switch, and statistics files.

Although the new file names will make configuration and support easier in the future and users were warned of the upcoming change in 8.9, this change will probably be the most traumatic 8.10 change for users upgrading from earlier versions.

**Beyond the MTA.** Outside the `sendmail` MTA itself, the open source distribution includes other utilities, such as `mail.local`, the local delivery agent; `makemap`, the map generation tool; and `praliases`, the tool that converts an alias database back to its textual form. These utilities have also had minor updates to improve ease of use. Although invisible to the end user, code sharing between `sendmail` and these utilities has increased by using portability and utility libraries. By sharing code and breaking the utility routines out of the MTA, 8.10 moves us a step closer to splitting up the monolithic `sendmail` process into multiple programs in future releases.

A new enhanced version of the `vacation` auto-responder is a standard part of the `sendmail` distribution beginning with 8.10. Some of the new features planned for the revised `vacation` include new command line options for specifying alternate databases and alternate messages, as well as a method for getting the sender

out of band. The new `vacation` will also support an exclusion list of addresses to which an automatic response should not be generated.

## 5.4. Maybe Next Time

As with any large software project, there are enhancements we had planned on including in 8.10 but were unable to tackle because of resource constraints. At the current time, the two biggest casualties were Windows NT<sup>TM</sup> portability and support for secure SMTP (i.e., encryption) [RFC2487]. While we continue to design with Windows NT portability in mind, the extensive changes required have lead us to postpone this change until a future version.

**Secure SMTP with TLS.** Although secure SMTP is an extremely important feature, arguably just as important as secure web service, the United States government is not expected to allow release of the source code for an encrypting mail server to the world. It is unfortunate that even though this encryption is widely available in other countries and freely available for download from international servers, the United States still has not recognized that the people being hurt most by these export restrictions on encryption are its own citizens and businesses.

Encryption patches for `sendmail` are available from one of a number of sites outside the United States. As an example, one can look at the `ssmail` patches at [http://www.home.aone.net.au/qualcomm/\[Rose99\]](http://www.home.aone.net.au/qualcomm/[Rose99]). However, this patch does not use the published TLS extension<sup>4</sup>.

We will continue to investigate methods of making secure SMTP with TLS available for `sendmail`. For example, we might produce a “domestic” version of `sendmail` with TLS.

## 6. The Future

There remain several major factors to research and goals to accomplish in future versions. `Sendmail` will need to move toward a threaded model to improve portability for Windows NT. This change will require significant changes to the MTA in both its use of global variables and memory management. Any services, such as DNS and system libraries, that `sendmail` uses

---

<sup>4</sup>This limitation may not be a disadvantage; the `ssmail` authors argue that the overhead of TLS is too high for routine use.

will also need to be thread safe. This change may improve or may degrade performance for UNIX systems depending on the thread implementation of the operating system and how it compares to forking, which has become quite efficient on some systems.

A popular trend in newer open source MTA implementations has been to break up the tasks into separate programs. We will be studying the performance trade-offs of making these changes to *sendmail* and breaking tasks off as appropriate. This approach has its benefits as “[i]t has been observed that one of the great successes of UNIX is that each tool does only one job, and therefore can do that job well” [Allm85]. It will also allow us to improve security by securing smaller portions of privileged code.

As we make *sendmail* portable to non-UNIX platforms, we will have to reconsider the I/O subsystem. For example, Windows NT and BeOS<sup>TM</sup> sockets do not have the same semantics as UNIX sockets. In particular, Windows NT does not have the *fdopen(3)* call, and BeOS sockets are not inherited by forked children. We expect to hide most of the I/O behind another compatibility layer, possibly *sfio* [Korn91].

After 8.10 is released, we expect to do considerable work on performance enhancements and tuning, including memory pools for more efficient memory allocation, support for threaded delivery, and the use of shared memory for saving long-term state.

## 7. Summary

As *sendmail* development continues, it is affected by four driving forces: continually changing network requirements, user requests, available development resources, and competition. None of these factors are particularly new to any popular network server software, but the substance of these factors for *sendmail* are unique. The paper has laid out these factors both historically and in the present, including some of the new features for 8.10 brought about by these four forces.

The most notable event in the evolution of *sendmail* development is undoubtedly the creation of Sendmail, Inc., a “hybrid” business model company producing both the open source and commercial versions of *sendmail*. Sendmail, Inc. helps drive network changes by participating more fully in the IETF. More directly, Sendmail, Inc. provides far more development resources—in the form of funding,

people, and tools—to the *sendmail* open source than were previously available. For example, the company has paid for conference calls between members of the Sendmail Consortium and plans to host meetings for the group.

This arrangement benefits both the company and the open source distribution. The open source gains new features and enhancements, while the commercial products reap the benefits of an active open source community contributing both new ideas and testing.

The future promises some exciting times for both the open source distribution of *sendmail* and the commercial products as both grow together.

The latest open source version of *sendmail* is available from <http://www.sendmail.org/>. More information about Sendmail, Inc. can be found at <http://www.sendmail.com/>.

## 8. Acknowledgments

The authors would like to thank David Conrad of the Internet Software Consortium, Sendmail, Inc.’s Mark Delany and Nick Christenson, and Kirk McKusick for their insights and suggestions regarding the paper. We also appreciate the hard work of Claus Assmann, Katie Calvert, and Cathe Ray, for reviewing and improving the paper. Finally, we would like to offer our sincere appreciation to the members of the Sendmail Consortium for all of their efforts and contributions to *sendmail*.

### REFERENCES

- [Allm85] Allman, Eric, and Miriam Amos, *Sendmail Revisited*, Proceedings of the Summer 1985 USENIX Conference, pp. 547-555, 1985.
- [Cost97] Costales, Bryan, and Eric Allman, *sendmail*. O’Reilly & Associates, Inc., Second Edition, 1993.
- [Korn91] Korn, David G., and Kiem-Phong Vo, *Sfio: Safe/Fast String/File IO*, Proceedings of the Summer 1991 USENIX Conference, pp. 235-256, 1991.
- [LASER] Freed, Ned, *LDAP Schema for E-mail Routing (laser) bof*, Meeting Report, Proceedings of the Forty-Third Internet Engineering Task Force, December, 1998.
- [Mark98] Markoff, John, *Battling Junk Email May Become Easier*, New York Times, March

17, 1998.

- [MSGFMT] Resnick, Peter W., *Internet Message Format Standard*, Internet Draft draft-ietf-drums-msg-fmt-07, January, 1999.
- [RFC821] Postel, Jonathan B., *Simple Mail Transfer Protocol*, RFC 821, August, 1982.
- [RFC1985] De Winter, Jack, *SMTP Service Extension for Remote Message Queue Starting*, RFC 1985, August, 1996.
- [RFC2034] Freed, Ned, *SMTP Service Extension for Returning Enhanced Error Codes*, RFC 2034, October, 1996.
- [RFC2222] Myers, John G., *Simple Authentication and Security Layer (SASL)*, RFC 2222, October, 1997.
- [RFC2476] Gellens, Randall, and John C. Klensin, *Message Submission*, RFC 2476, December, 1998.
- [RFC2487] Hoffman, Paul, *SMTP Service Extension for Secure SMTP over TLS*, RFC 2487, January, 1999.
- [RFC2505] Lindberg, Gunnar, *Anti-Spam Recommendations for SMTP MTAs*, RFC 2505, February, 1999.
- [RFC2553] Gilligan, Robert E., Susan Thomson, Jim Bound, and W. Richard Stevens, *Basic Socket Interface Extensions for IPv6*, March, 1999.
- [RFC2554] Myers, John G., *SMTP Service Extension for Authentication*, RFC 2554, March, 1999.
- [Rose99] Bentley, Damien, Greg Rose, and Tara Whalen, *ssmail: Opportunistic Encryption in sendmail*, Draft.
- [Salz92] Salz, Rich, *InterNetNews: Usenet transport for Internet sites*, Proceedings of the Summer 1992 USENIX Conference, pp. 93-98, June 1992.
- [SMTPUPD] Klensin, John C., *Simple Mail Transfer Protocol*, Internet Draft draft-ietf-drums-smtpupd-10, February, 1999.

# Meta — a freely available scalable MTA

Assar Westerlund

*Swedish Institute of Computer Science*

assar@sics.se

Love Hörnquist-Åstrand

*Department of Signals, Sensors, and Systems, KTH*

lha@s3.kth.se

Johan Danielsson

*Center for Parallel Computers, KTH*

joda@pdc.kth.se

## Abstract

This paper describes the design and implementation of the mail server Meta. It is intended to be a simple and secure yet efficient and scalable mail server. It only handles receiving mails by SMTP and providing a POP server for the user but tries to be fast at doing that.

## 1 Introduction

One of the oldest and still most popular applications on the Internet is electronic mail (e-mail). More and more people find e-mail to be a practical form of communication with colleagues, companies, authorities, relatives, and friends. For many people, e-mail has become the preferred means of communication. And there are lots of mailing lists where people interested in a particular topic can discuss it among themselves. Also, people are depending more on e-mail actually working and getting to the recipient in a short amount of time.

Lots of effort has been put into optimising other common Internet applications, like web servers and proxies, news servers, and others. But few have looked at how to handle large volumes of mail for large number of users efficiently. We think this popular application deserve some more attention.

## 2 What is the Problem?

The problem that is addressed (and/or solved) by Meta is building a high-capacity, scalable, and secure mail-hub. A mail-hub in this context is a server that receives mail destined for users with the Simple Message Transfer Protocol (SMTP)[11] from the Internet and that allow them to retrieve it with the Post Office Protocol (POP)[12] to their local mail clients. In other words, Meta is an embedded

SMTP-server and POP-server. This is shown in figure 1. Meta does not try to be a general-purpose MTA. Instead, it tries to fulfil the particular need that we had and doing it efficiently and while remaining simple and secure. While performing a subset of what a general-purpose MTA does, we think that there are a number of sites other than us that require this functionality, such as companies or universities with large number of mail users and above all, large ISPs.

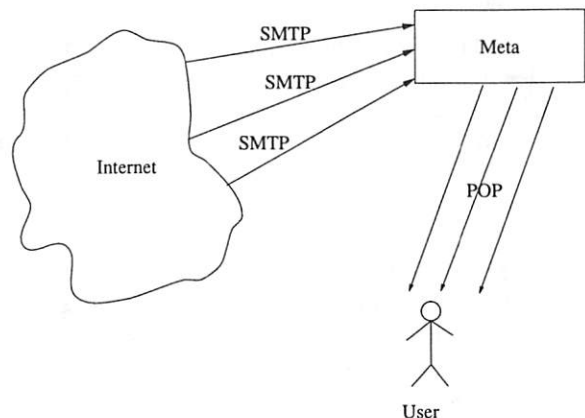


Figure 1: Meta overview

The traditional way of building a mail hub is to run an MTA such as sendmail[10] that receive mails. They are then written in spool files by a local delivery program such as mail.local. There exists a single flat file per user where the incoming mails are stored. Each new mail is appended to the end of this file. The mails are read from these files and eventually deleted by the pop daemon when the users fetch them to their mail clients with POP. But some users and mail clients instead access the spool files directly on the mail hub or through NFS [13]. Or they use IMAP [14] or some other (and new) means of getting the mail. This

means that the mails have to be stored in the common mbox-format so that all the programs that can access them will understand the format.

With Meta we decided early that the only way of accessing the mail would be with POP. This simplifies lots of things and does not require storing mail in the mbox format. (The benefits from this are quite similar to those described in [7] regarding NNTP [15] and news spool files.)

### 3 Goals

- little (or no) configuration

We would like Meta do be as self-functioning as possible. This means not having to maintain complicated configurations and/or do lots of tuning on the running Meta. The configuration is now kept quite minimal.

- focus on solving a small and well-defined problem

This is a traditional UNIX approach. Meta itself doesn't relay mail, handle outgoing mail, or manage mailing lists. These functions, which are of course needed, will be implemented in separate programs. Separating different functionality into different programs tend to make them simpler and more efficient.

- simple

The MTA should be simple to make it easy to review for correctness, and to simplify a security-audit. Lesser code makes it easier to write correct code. It should also be simple to extend and give new functionality.

- secure

An MTA is exposed to a large number of more or less broken mail applications, as well as malicious people trying to do evil things, but people trust mail to function and depend on it, so it is rather important that it is reliable and secure.

- efficient

It should be possible to handle a large number of users receiving very large number of mails on rather low-end hardware.

- scalable

Scalability is an important point, it should not be necessary to buy a faster computer just to be able to handle more users, instead you should be able to cluster mail servers to handle the additional load.

### 4 Non-Goals

- being a complete MTA

Which means not handling UUCP, address rewriting, etc, etc.

- being configurable in every detail

There is not going to be a sendmail.cf to configure Meta. Its sole purpose is to be able to receive mail really fast.

- being compatible with old MTAs

The important point is using the protocols correctly, the mail hub itself should be a rather dark box. Not having to be compatible with old ways of doing things (like /var/spool/mail/user and .forward) means being able to try to solve the problems in new (and hopefully better) ways.

- sending and relaying mail

Separate programs will be written to handle these tasks. They do not even have to run on the same servers.

- handling mailing lists

There are already lots of programs that do this so we have not found any need for writing a new one.

### 5 What is the Limit?

We wanted to get a feeling for how good performance it would be possible to get out of a mail server running on a common piece of hardware, say an ordinary PC. For this purpose, we designed a simple benchmark for measuring the time it would take to receive a single mail over an SMTP connection, and also wrote a simple prototype of a mail server. This mail server just receives mail and appends it to spool files, one for each user. The file is also locked at the beginning of reception and unlocked at the end. After having written the complete mail, the file is fsync'ed and close'd. The existence of a user is determined from the existence of a spool file. It uses a state-machine written around select.

The benchmark program sent a variable number of mails of around 500 bytes to one of 10 different users in a round robin fashion over the same SMTP connection. We measured the total elapsed time for this operation.

To get a rough comparison with other MTAs, we also performed the same benchmark against sendmail 8.8.7. We did not spend any time optimising the configuration of the sendmail but just used a stock configuration file. We were not after a heads-to-heads comparison but rather a simple estimate of how Meta compared with sendmail. Both Meta and sendmail ran on a 200 Mhz Pentium Pro with 48Mb of memory, an IDE disk, and under FreeBSD 2.2.

The first tweak we made on our SMTP server was to remove the call to fsync to see how the speed would change with not having to perform synchronous disk writes. As can be seen in figure 2 the change is quite dramatic. From around 200 mails/second it increases to around 1000 mails/second. And the slope of the curve is smaller a well.

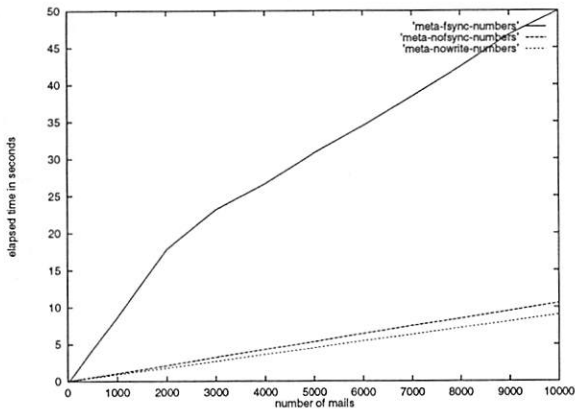


Figure 2: Comparison sync, async, and no-write mail reception

Actually, the extreme case would be to remove the disk I/O completely to see how much of a bottleneck the I/O is. In figure 2 it can be seen that the performance when using asynchronous writes is quite close to that of not having any disk activity at all, and quite far from the synchronous case. Compare the asynchronous case with the no I/O case in this figure and note how much more time the benchmark takes when the writes are synchronous.

A possible optimisation at the SMTP protocol level is to have the mail generator do SMTP pipelining [16]. This means it does not have to wait for every reply to every command before sending the next one and increases the throughput, above all when the sender is fair away. Even in our benchmark case where the sender is very close it makes a difference as can be seen in figure 3. In practice, the savings from doing pipelining is largest when a lot of mails are sent from the same host.

In summary, in table 1 are the approximate number of mails per second it's possible to receive with different configuration, including the one running sendmail.

The conclusions from this experiment was that fsync is going to be important performance-wise and that it is possible to write something that achieves quite a lot better performance than sendmail.

How can we avoid the penalty associated with fsync?

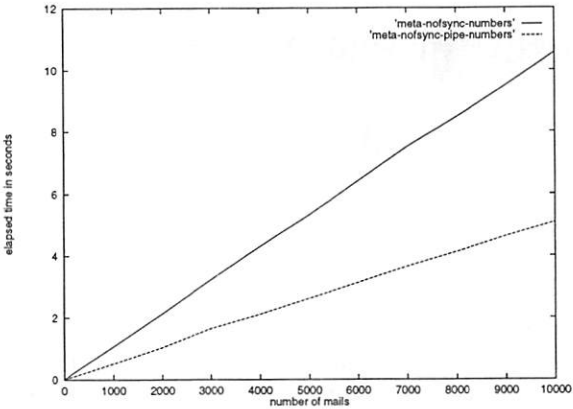


Figure 3: Comparison no-pipelining and pipelining

description	mails/second
meta-fsync	~ 200
meta-nofsync	~ 1000
meta-nowrite	~ 1200
meta-nofsync-pipe	~ 2000
sendmail-8.8.7	2 – 10

Table 1: Receiving rates for different configurations

- do not fsync  
That is not a satisfactory solution because when you acknowledge have received a mail you are not supposed to be able to loose it. Users do not like dropped mails.
- fork a different process and let it fsync  
That might be doable but it would require the overhead of fork instead of that of fsync.
- send the file descriptor to a running process  
That seems like one of the best ways of resolving the problem
- clustering  
Have two or more nodes store copies of the mail. With enough nodes running with UPSes storing the mail we might say the system is reliable enough. Clustering is discussed more in section 10.

## 6 Mail Storage

Meta stores the incoming mails in a series of fix-sized logs. Each message is appended at the tail of a log. When writing the mail is complete, an entry is written for every recipient describing where to find it and in which log. A reference

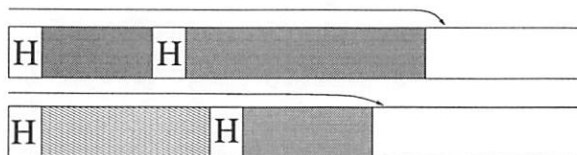


Figure 4: The mail logs

counter is also kept for each message. It is set to the number of recipients when storing the mail and is decremented every time an user deletes the mail in a POP session. As the mails are stored consecutively in the logs, only one mail can be written to the same log at any time. The log is locked while writing the mail so that there can only be as many concurrent SMTP sessions as there are logs. On the other hand, the same user can receive several mails at the same time without causing any waiting for locks. When the storage space is too full or Meta is idle, a garbage collector is run that will copy the non-deleted mails from a full log to new logs. To avoid unnecessary coping of the message when it is garbage-collected several times, a generational garbage-collecting algorithm is used. A picture of the logs is shown in figure 4. The arrows point to the first free position in the corresponding log.

The assumptions that we base this choice of data structure on is that mails stay around for a short period of time and are then picked up the client. If the goal was to support something like IMAP where the mails are stored for a long time, some other organisation would probably be better.

Because all mails are received over SMTP and fetched over POP, both of which has the same formatting and quoting rules (CR+LF as line terminator, a single dot on a line as message terminator), the messages are stored in the wire format and are never converted. The CNFS storage for INN can also be configured to behave this way, see [7].

## 7 Security

Meta runs as an unprivileged user in a chrooted directory. Because no users need to access the spool files, their permissions and owners are not important. All files are owned by the 'meta' user and only read and writable by this user. The mail users need not have accounts and/or shells on the mail hub and in fact that is the recommended configuration. It also makes for a more stable solution as the users have less opportunities to endanger the stability or performance of the mail hub.

The only operation that cannot be done as the meta user is binding the smtp and pop ports. That is done by a small 'nanny' program that just binds the ports, does a chroot to

the meta directory, changes the uid, and starts the meta program itself. The nanny also is responsible for re-spawning meta. It is a simple program and consists of only ~ 70 lines of code and should be easy to audit. That is the only code that run as root. On an operating system that is not that paranoid about ports < 1024 it is not needed either. No program has any set[ug]id bit.

Trying to keep it simple by having it just solve one well-defined problem and not doing any complex stuff should also help making it more secure.

The security model used by Meta is quite different from recent MTAs that have focused on security, typically qmail [9] and postfix [8]. Both of them are composed of a collection of small programs that communicate through IPC and the file systems and that do not trust each other. Meta is a "monolithic" program that handles all in the same program and process.

## 8 Layers and "back-ends"

Meta has a well-defined API to allow new back ends (called layers) to be added that receive (and store) mails differently. Currently the above described log-based layer, a layer that only sorts mails, and a null layer have been implemented.

## 9 User Database

Meta keeps a database of all its users. This is completely different and separate from the ordinary `/etc/passwd`. The database can however easily be generated from a standard passwd-file. The database contains the complete mail addresses of the users, including the domain part. This allows Meta to do a lookup on the complete address when checking if it should accept mail for a user. Only if the entire address exists in the database is the mail accepted. It is therefore unnecessary to configure for what domains Meta should receive mail.

Other per-user information for features that we have not implemented yet (mail quotas, spam filtering, ...) will also be stored in the user database.

## 10 Clustering

While it should be possible for large mail-hubs to run a single Meta instance on a single machine, the reliability might not be as high as needed and there might not be room to grow the mail volume gracefully. Therefore we are in the process of implementing support in Meta for running a cluster consisting of collaborating Meta daemons on different machines. We assume that all of them will be presented as SMTP-servers to the Internet at large and to the

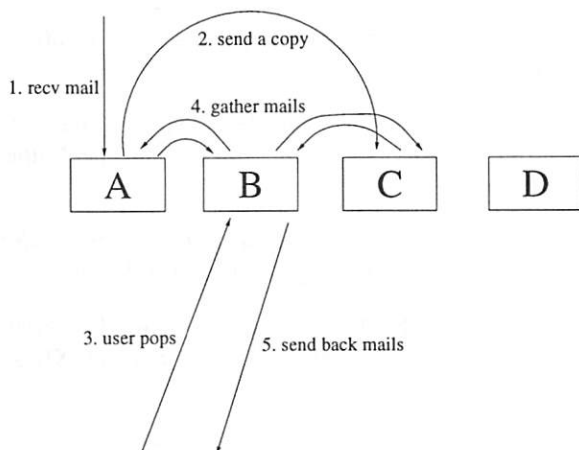


Figure 5: Mail cluster

POP users and therefore be identical. Also, the bandwidth between is considered to be plentiful. The goal here is that Meta should scale, that is, it should be possible to handle more users and larger number of mails by just adding more machines. These machines should share the load and be configured as part of the cluster easily. Meta also tries to store the mails redundantly so that no mails are lost should a single machine crash. The users need not keep tracking of on which machine they have their mails stored but are able to fetch them from any node in the cluster.

A typical scenario is shown in figure 5. First in step (1), the sender of the mail looks up MX records for the Meta cluster. In this case, the first host returned is A. A SMTP connection is opened to A which will while receiving the mail (2) send a copy of it to C. At some time later, a user tries to pop her mail and looks up an A record, getting B. In (3) she starts popping from B. B will then retrieve the mails stored on A and C in (4) and in (5) send them to the user.

An obvious question is how to choose the other server that will store the redundant copy of a mail. Two policies seem the most appropriate at the moment:

- trying to keep all the mails belonging to a single user on the same host. That way all messages will be stored on the machine that received the mail and the “home” machine for the user.
- choose the least loaded machine, which should give reasonable load-sharing over the nodes in the cluster.

To get some idea as to how much sending an extra copy of the incoming mails to another server would cost, we added that function to the server used in section 5. As is shown in figure 6, the overhead is quite small (around 15%).

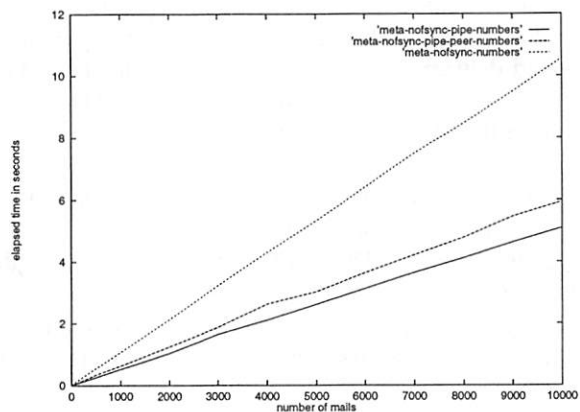


Figure 6: Overhead of sending a copy to a peer

## 10.1 Load Balancing

The load-sharing could be performed by having multiple A records, a load-balancing name server ([17]), or some kind of TCP router ([18, 19]). We have chosen to start with the simplest solution (DNS round-robin) and see if that gives good enough load balance.

## 11 Usage

Meta is not quite ready to be used in production yet but there are some sites running it experimentally and it is being installed as a front-end to the mail system at a large Swedish ISP. Meta was chosen for its performance at receiving mail.

## 12 Related Work

Christenson [3] tries to make a scalable mail solution using file servers (NFS) to distribute the mail storage (and thus vulnerable to network-partition). The smtp and pop hosts are not keeping any local data. The local delivery agent is replaced but otherwise a stock sendmail is used. Meta tries to solve a similar problem but from scratch instead of building on sendmail.

Carson[4] discusses the security paradigm “least privilege” applied on mailing. Tries to solve the problem with giving sendmail access to port 25 (SMTP/TCP) by adding a wrapper to sendmail that does that. Meta does exactly that, but do the delivery to the mailbox itself instead of using a local delivery agent.

Knowles[5] focuses on transport issues (inbound, mostly outbound, mailing lists). Arguing that the mail queueing is the slow part of mail delivery, showing techniques to speed-up sendmail’s delivery time. Kolstad[6] also tries to tune

sendmail to deliver mail faster for mailing lists. Meta is not today trying to solve that problem, it is trying to make in-bound delivery as fast as possible. A outbound MTA could use Meta's logs to avoid most of the problems with the queue-files described in the paper.

Porcupine[1, 2] is architecturely very similar to Meta. They also build a cluster of identical server machines that act as a large mail server. The storage of mails locally on the nodes is however done differently from Meta. The next stage of the Porcupine project underway now is aimed at generalising the ideas that were used for building their mail server.

### 13 Future Work

There a more work to be done with regards to clustering, to see what are good policies for how to choose server nodes and when to migrate mails between nodes. Load balancing also requires some experimentation and measurements.

There are functionality that we have not implemented but is probably going to be needed like mail quotas, filtering per user (for spam), and other related functionality.

### 14 More Information

See  
<http://www.stacken.kth.se/projekt/meta>.

### 15 Acknowledgements

Björn Grönvall was responsible for a large part of the initial ideas that led to Meta. Magnus Ahlertorp has written part of the code and has also participated in the discussions about Meta.

### References

- [1] Yasushi Saito, Brian Bershad, Hank Levy, and Eric Hoffman, *The Porcupine Scalable Mail Server*, SIGOPS European Workshop, Sintra, Portugal. September, 1998.
- [2] David Becker, David Becker, Brian Bershad, Bertil Folliot, Eric Hoffman, Hank Levy, and Yasushi Saito, *The Porcupine Project*, <http://www.porcupine.cs.washington.edu/>
- [3] Nick Christenson Tim Bosserman, and David Beckemeyer, *A Highly Scalable Electronic Mail Service Using Open Systems*, USENIX Symposium on Internet Technologies and Systems, Monterey California (1997)

- [4] Mark E. Carson, *Sendmail without the Superuser*, 4th UNIX Security Symposium, Santa Clara California (1993)
- [5] Brad Knowles, *Sendmail Performance Tuning for Large Systems*, SANE98, Maastricht, The Netherlands (1998)
- [6] Rob Kolstad, *Tuning Sendmail for Large Mailing Lists*, LISA97, San Diego, California (1997)
- [7] Scott Lystig Frichie, *The Cyclie News Filesystem: Getting INN To Do More With Less*, LISA XI, San Diego, CA (1997)
- [8] Wietse Venema, *Wietse's Postfix Project*, <http://www.postfix.org/>
- [9] Dan Bernstein, *qmail: a replacement for sendmail*, <http://www.qmail.org/>
- [10] Eric Allman, *Sendmail*, <http://www.sendmail.org/>, <http://www.sendmail.com/>
- [11] Jonathan B. Postel, *SIMPLE MAIL TRANSFER PROTOCOL*, Information Sciences Institute, University of Southern California (1982)
- [12] J. Myers, M. Rose *Post Office Protocol - Version 3* Carnegie Mellon, Dover Beach Consulting, Inc. (1996)
- [13] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, B. Lyon, *Design and Implementation of the Sun Network Filesystem*, In Proceedings of the USENIX Summer Technical Conference, 1985.
- [14] M. Crispin, *Internet Message Access Protocol - Version 4rev1*, University of Washington (1996)
- [15] Brian Kantor, Phil Lapsley, *Network News Transfer Protocol*, U.C. San Diego and U.C. Berkeley (1986)
- [16] N. Freed, *SMTP Service Extension for Command Pipelining*, Innosoft (1997)
- [17] Roland J. Schemers, III, *lbnamed: A Load Balancing Name Server in Perl*, LISA IX, Monterey, CA (1995)
- [18] Cisco, *Cisco LocalDirector*, <http://www.cisco.com/warp/public/cc/cisco/mkt/scale/locald/index.shtml>
- [19] IBM, *SecureWay Network Dispatcher*, <http://www.software.ibm.com/network/dispatcher/>

# Porting Kernel Code to Four BSDs and Linux

Craig Metz

*ITT Systems and Sciences Corporation*

`cmetz@inner.net`

## Abstract

The U.S. Naval Research Laboratory develops and maintains a freely available IPv6 and IP Security distribution. All of the software builds and runs on BSD/OS, FreeBSD, NetBSD, and OpenBSD, and a growing portion of the software builds and runs on Linux. Each of the four BSDs has evolved significantly from their original 4.4BSD-Lite ancestor, and increasingly more of that evolution is along divergent paths. Linux shares no significant ancestry with the BSDs, but is still a POSIX system, which means that many of the same high-level facilities are available even though their implementation might be completely different.

This paper discusses many of the differences and many of the similarities we encountered in the internals of these systems. It also discusses the techniques and glue software that we developed for isolating and abstracting the differences so that we could build a significant base of system code that is portable between all five systems.

## 1 Introduction

### 1.1 History

The U.S. Naval Research Laboratory's IPv6+IPsec distribution [1] began its life in 1994 on BSD/386 1.1, which was a 4.3BSD "Net/2" system. When BSD/386 2.0, which was based on 4.4BSD-Lite, became available, we moved the code to that system. This was a straightforward change, as the differences in the parts of the two systems that our code interfaced with were small. We then added support for "real" 4.4BSD/sparc. At this point, we ran into our first two experiences with maintaining the same code in different kernels. First, we had to separate our code into parts specific to each and parts shared between the two. Second, we found slight differences in the way the systems did things and had to add some `ifdef` statements to our "shared" tree to cope with these. Still,

these were virtually identical systems. Because a large portion of our code resided in the `netinet` directory and the two systems were virtually identical, we decided to put our modifications into the 4.4BSD-Lite2 `netinet` (IPv4) code and make the needed changes to port that into BSD/386 and 4.4BSD/sparc.

In 1995, we made the first jump to a radically different system. Although most of the research community used BSD, Linux was also an interesting system, so we decided to attempt to port a component of our software to Linux. One component that was somewhat unique to the NRL software was our PF\_KEY[2] interface and implementation, which also had the important feature of being a fairly self-contained module. PF\_KEY communicates with user space as a new sockets protocol family and with the rest of the kernel through function calls we defined. We felt that it would be useful to have the PF\_KEY implementation running on Linux even if the rest of our code never did, and that it was a reasonable part of the software to attempt to port. We split our PF\_KEY implementation and our debugging framework into a OS-specific parts (such as the sockets interface code) and common parts (such as the actual message processing code). With some help from the Linux community and Alan Cox in particular, we ported our 4.4BSD PF\_KEY implementation to Linux in about three months.

In 1996, we added support for NetBSD, another 4.4BSD-Lite derived system. The NetBSD team had made a number of changes versus 4.4BSD-Lite that we had to add `ifdefs` to handle. NetBSD had also added a number of new IPv4 features that 4.4BSD-Lite didn't have; because we were using a common 4.4BSD-Lite `netinet` implementation, those features were all removed when we replaced that code.

As time went on, we dropped support for the "real" 4.4BSD/sparc because almost nobody actually had a copy of it and because NetBSD/sparc was a freely available system that worked better. Both BSD/386 (renamed to BSD/OS) and NetBSD continued to

evolve, and we had to put more and more `ifdefs` into our code to cope with their changes from 4.4BSD. It became clear that we were actually spending almost as much effort shoehorning the original 4.4BSD-Lite `netinet` code into the newer version of these systems as it would take to maintain our changes to the `netinet` code in each system's native implementation, and throwing away the native systems' changes was a problem for some users.

In 1997, we built an implementation of the second version of our PF\_KEY interface. We chose to build this implementation as a ground-up rewrite. Instead of building this new implementation on one system and "porting" to the others, we actually developed this code simultaneously on BSD/OS, NetBSD, and Linux. By doing this, we were able to find and resolve OS dependencies quickly, and we were able to take advantage of the debugging strengths of each of these systems. This was also an opportunity to do many things differently, designing for portability based on our previous experiences rather than adding portability as an afterthought.

In late 1997, BSDI chose to integrate our code into their source tree for the upcoming BSD/OS 4.0. BSDI actually did the work of integrating our changes into their `netinet` code, which meant that half of the work of integrating our code into the native `netinet` code of our supported systems was done. We also decided to expend more effort on becoming a good choice for integration into other systems, which meant adding new ports to OpenBSD 2.3 and FreeBSD 3.0 and integrating our changes into the native `netinet` implementations of each system.

The OpenBSD port was surprisingly straightforward because of its shared ancestry with NetBSD; most of the differences between 4.4BSD-Lite and NetBSD could also be found in OpenBSD. The majority of the work we ended up doing for this port was integrating our code into the OpenBSD `netinet` code, which then got us most of the way to also having our code integrated into the NetBSD `netinet` code.

The other port we added was to a snapshot of FreeBSD 3.0. This port was much more difficult, because FreeBSD had made many more substantial changes from 4.4BSD-Lite than the other BSD systems we supported. Working with a snapshot turned out to create problems, too. The system had bugs that we ran into, and many of the system's debugging facilities didn't work at all. We also had more work to do when we moved from the snapshot to the real 3.0 release, as there were significant new changes versus 4.4BSD-Lite that we had to handle.

As both the OpenBSD and FreeBSD ports were starting, we had some level of support for five operating systems and we were maintaining much more

OS-specific code than we were before because we were modifying each of the BSD systems' native `netinet` code. We decided to do a major reorganization of our source tree, separating it into seven modules: an OS-specific piece for each of the five systems, a BSD-common piece shared between all of the BSD systems, and a common piece shared between all of the systems. With some added scripts and tools, we were able to create an organization of "overlays" such that the common pieces could be symlinked to appear in the OS-specific directories, and such that the more-specific pieces could override the less-specific pieces. This new organization made it much easier for us to maintain our software on all of these systems while trying to balance duplication of effort with close integration into the supported systems.

While we were doing the ports to OpenBSD and FreeBSD, we were also rewriting large portions of our IPsec implementation. Because we knew that it was a direction we wanted to eventually move to, we built a framework to allow the new code to be ported to Linux. After all of the BSD ports and the rewrite of the IPsec code were finished and released, we began the task of porting our IPsec code to Linux, which is currently a work in progress.

## 1.2 Observations

In the course of building our software and adding support for all of these systems, we learned many things about how to build and maintain portable kernel code. Many of the lessons that we learned didn't have to be learned the hard way. Large blocks of code are shared between the various BSD systems, especially code for new networking features and device drivers. There are also some cases of software that had been ported from BSD to Linux (for example, the NCR SCSI driver) and from Linux to BSD (for example, the GPL x87 math emulator). Perhaps the best known example of porting kernel code is the BSD network stack, which can be seen running in all sorts of systems that are definitely not BSD UNIX. However, we are not aware of anyone who has ported code to the various BSD and Linux systems and really documented what they discovered while doing it.

The rest of this paper has two major parts. In the first part, we will document our high-level discoveries about building and maintaining portable kernel code. These are general observations about how to structure such code, how to go about building such code, and – possibly more important – what we discovered that you should *not* do. In the second part, we will document many of the specific mechanisms we developed for making code portable. These range from simple wrapper macros to major new compatibility data structures.

## 2 High-Level Discoveries

### 2.1 Should You Port It?

Probably the most important thing to consider before getting to far into thinking about porting kernel code is whether something really should or shouldn't be ported.

Given enough time and effort, any piece of system code can be ported to any other system. However, for many pieces of code, that will mean either porting a large chunk of the original OS with it or otherwise dramatically changing the target OS to be more like the original OS. Generally speaking, this is not a good thing to do.

Some code is so dependent on the system that the problem that the code solves might not exist in other systems or the solution approach might not work in other systems. Or maybe someone else is already doing a good enough job of solving the problem on another system that you don't need to port your code there.

This is a lesson we learned, though not the hard way. While we were getting our IPv6 implementation for 4.4BSD and NetBSD more stable, Pedro Marques and a few other developers were working on an IPv6 implementation for Linux. An IPv6 kernel implementation requires extensive changes to the existing system to generalize IPv4-dependent code, and these changes are very system dependent. Since that represents the majority of the effort required to build an IPv6 implementation, it didn't make sense for us to port our kernel implementation to Linux, since we would have to do most of the work from scratch only to be duplicating the effort of another group.

Note also that some systems are architecturally similar and some are very different, and it makes a lot less sense to try to port kernel code between very different systems than very similar systems. For example, both 4.4BSD and Linux are UNIX-like systems, and they both have sockets-based IP network stacks. So, while the implementation of network code might be very different between the two systems, there are still a lot of high-level similarities because they are constrained in their architectures by the standardized interfaces they present (POSIX, sockets, IP). In contrast, porting code from the Linux kernel network stack to the Win95 kernel network stack might be a lost cause because the systems don't share enough architectural similarities.

Another important note is to observe the distinction between kernel space and user space. While it is sometimes possible to take a module written for kernel space and move it to user space or vice versa, most code written for use in kernel space is that way for good reasons. As a general rule, don't try to move code through the kernel/user barrier.

### 2.2 Portability Techniques

Writing really portable code is not hard, but it is tricky. Almost all of the same approaches, tricks, and traps that you encounter porting user-space code applies to kernel-space code, too. One of the unfortunate problems resulting from increased standardization of systems is that a lot of people have never ported code from one radically different system to another, so many people really don't know how to take code and make it portable.

As a general rule, a good approach to making code portable is to add new abstractions. For substantially similar operations that just work a little differently on different systems, this approach works really well – replace the concrete code with an abstract macro that expands into different code on different systems. You might also consider giving up some of the flexibility that particular systems give you. For example, BSD systems have a function `malloc(size, type, wait)`, while Linux has a function `kmalloc(size, flags)`. We abstracted these into a single `OSDEP_MALLOC(size)` function, which does what we really need each to do.

We created a lot of `OSDEP_x` macros to abstract away minor system-dependencies in our code, and we found that this approach works very well for small differences. Another common approach to the same problem is to use conditionals around blocks of code, and to provide an alternative for each system. Figure 1 shows an example of the code that results from this approach. For small differences, we believe that abstracting leads to more readable code and makes it harder for system-specific code to get out of sync.

Abstracting is also far more convenient for debugging than conditionals. The code in Figure 1a might expand to end up the same as the code in Figure 1b. But, when built with debugging enabled, `OSDEP_MALLOC()` actually gets defined as a call to our `malloc()` debugger code, and `OSDEP_RETURN_ERROR()` actually gets defined as a set of statements that tell us what line threw the error. Changing the code in this way is easy to do with macro abstractions, but would be painful to do with code surrounded by conditionals.

One of the serious dangers of conditionals is the temptation to make them either-or statements. For example, our earlier code contained conditionals that evaluated to one block on Linux systems and another block on non-Linux systems. This hard-codes a very dangerous assumption: that you won't be adding new ports to the mix that will make life less simple. We have been bitten a number of times by conditionals of this form when adding new ports, and we learned the hard way to be very careful with the preprocessor's `else` directive.

An important side note is that macros are strongly preferable to functions in kernel space. For applica-

a.

```
if (!(pfkeyv2_socket = OSDEP_MALLOC(sizeof(struct pfkeyv2_socket))))
    OSDEP_RETURN_ERROR(ENOMEM);
```

b.

```
#if __bsdi__ || __NetBSD__ || __OpenBSD__ || __FreeBSD__
    if (!(pfkeyv2_socket = malloc(sizeof(struct pfkeyv2_socket), M_TEMP, M_DONTWAIT)))
        return ENOMEM;
#endif /* __bsdi__ || __NetBSD__ || __OpenBSD__ || __FreeBSD__ */
#if __linux__
    if (!(pfkeyv2_socket = kmalloc(sizeof(struct pfkeyv2_socket), GFP_ATOMIC)))
        return -ENOMEM;
#endif /* __linux__ */
```

Figure 1: Two Approaches to Minor Differences: (a) abstracting (b) conditionalizing

tion code, the overhead of a function call is a small price to pay for compatibility. In kernel space, performance and memory usage (both in terms of code size and stack use) is much more important, and so adding function calls that contain little code is probably a bad thing to do. In the more general sense, whether to put things in functions or to inline them as macros is still a judgment call the programmer has to make.

For larger differences, abstraction takes much different form. There, large functions or sets of functions, all surrounded by conditionals, is probably a reasonable approach. For example, a large part of our PF\_KEY implementation is the interface between our messaging code and the system's socket layer. The different systems' socket layers required radically different interface functions and data structures to be provided. Since the organization and structure of what had to be done varied so much between the systems, we provided separate versions of these functions for Linux and for the BSDs, with more conditionals surrounding some of the differences among the BSD functions. However, the sockets interface code provides a uniform interface to the rest of our PF\_KEY code, so the same messaging code can send up a message through the Linux sockets layer, the FreeBSD sockets layer, or the BSDI sockets layer, and the messages have essentially the same format.

Another large difference that needs abstraction is significant data structures. BSD systems use a chain of `struct mbufs` to hold the contents of packets, while Linux systems use `struct sk_buffs`. These are very different structures, so abstraction of significant accesses to these structures won't work... at least, not without either making assumptions or serious performance degradation. In our PF\_KEY code, we were relatively lucky in that we were able to simply define macro functions that copy blocks of data into and out of the systems' native buffers and were done.

In our IPsec code, however, we were not nearly so lucky – now we need to take a packet in the native buffer, operate on it, and return a result in the native buffer, and doing that by copies is not reasonable because it uses too much extra memory and costs too much for acceptable performance. So we defined an abstract buffer structure – the `struct nbuf` (see Section 3.3 for more details) – and defined “border functions” that take a native buffer and turn it into a `nbuf`, or take a `nbuf` and turn it into a native buffer. Those border functions are designed to use certain tricks and to check for useful cases that avoid copying the actual buffer contents in the common case, so the cost of using the abstract `nbuf` rather than the system-native structure is only that of the `nbuf` header itself and the cost of going through the border functions, both of which are small costs. The benefit is, in our opinion, huge. We now have a uniform, reasonable buffer that we can work with regardless of the system and a set of known properties of that buffer we can use to write simpler and faster code.

Which portability approach to take is basically a trade-off, and knowing which of the available options to take is basically a matter of experience and intuition. There is almost always more than one way that you can do it, and you can always go fix things later if you decide that you made the wrong choice.

## 2.3 Debugging

Each of the five systems that we support has different debugging capabilities. Many of them have the same facilities available (e.g., `kgdb`, `kdebug`, `ddb`, core dumps, display of trap information), but the reliability and net utility of those functions varies dramatically from system to system and among hardware platforms on the same system. I've never met an experienced kernel programmer who won't admit that all kernel de-

bugging facilities are at best a mixed blessing: handy when they work, but they don't always work when you need them to. When your code has bugs that go trashing things in kernel space, it's not too hard for it to trash things that debugging facility needs (or the debugging facility itself, for that matter!).

There are several rather immortalized Linus Torvalds quotes about kernel debugging, but the summary of them is pretty simple and exactly agrees with our experience: The best approach to debugging kernel code is to read it, and read it carefully. Debuggers are a wonderful thing, but they tend to entice you into an interactive mode of programming where more time is spent trying to figure out if the code you have does the right thing than trying to figure out if the code you have is *right*. Especially when you're running on systems you just ported your code to and aren't as experienced with, reading your code and reading the system-native code you call is a useful thing to do when you're having trouble.

Though it's certainly a lot less convenient way to do things, we've found that the one relatively constant thing across the kernels we support is good old `printf()` (well, Linux calls it `printk()`, but a simple preprocessor `define` statement solves that problem). With only one exception (4.4BSD/sparc at high software priority levels), you can always use it to print data out to a console. By inserting `printf` statements in interesting places, you can binary search for the trouble spot and display the contents of variables that are interesting to the trouble code. This is a lot slower way to do things than attaching a debugger and single-stepping, but there are a lot fewer things that can go wrong and it seems to work on all systems. Using `printf()` also tends to change the execution properties of code being observed much less than kernel debuggers, which is important for certain classes of bugs. Code that starts working fine when under a debugger is not fun to debug.

Another set of tools that are commonly available and handy are the object tools. One technique we use all the time is to take the instruction pointer and stack contents from a trap, run `nm | sort -n` on the kernel binary to get the addresses of functions and data structures, and to figure out what function was executing, what functions called it, and what global data structures it was working with. If you built a kernel with debugging symbols, more detailed execution information can be gotten by running `gdb` on the image, listing the function you found was executing, and using the `info line` command to binary search for the actual line of code. Systems with the GNU toolchain have a command `addr2line` that does this for you, which is quite handy. Note that trap information can be misleading for certain types of bugs, so be suspi-

cious if the information you get doesn't make sense. For example, we have found bugs where code trashes the stack frame and the function's return will cause execution to jump off into space; the trap address in that case can be all sorts of interesting values, none of which tell you where the bug is.

### 3 Detailed Discoveries

#### 3.1 Differences Between BSDs

BSD/OS, FreeBSD, NetBSD, and OpenBSD are all derivatives of the 4.4BSD-Lite released from the University of California, Berkeley (most if not all of them have been updated to incorporate the patches in 4.4BSD-Lite2). In this paper, I don't have enough space to do justice to these systems' evolutions beyond this common ancestry, but I think it's important to describe some of the differences that affected our code. These are almost all local to the network stack.

In my opinion, BSD/OS has remained the closest to the shared ancestor, followed by OpenBSD and then NetBSD, with FreeBSD most aggressively changing from the common base. Many of the changes that the systems have made are not clearly good or bad, but are design trade-offs. The same set of changes is frequently considered evolution by some and devolution by others. From the point of view of portability, however, extraneous differences are bad because they require more abstractions or conditionals. In discussions with some of the systems' maintainers, this is not considered to be a problem, because portability of kernel code is not a priority. Hopefully, the maintainers of systems will reconsider this in the future.

One of the most mundane yet more prevalent changes the systems made is to make linked lists use the BSD `sys/queue.h` `TAILQ` macros rather than each having different implementations. Between the four BSD systems we support, there are three different ways this ended up getting done. For example, the interface address lists (`struct ifaddr`) are done in FreeBSD as `TAILQs` in with a field named `ifa_link`, NetBSD and OpenBSD as `TAILQs` with a field named `ifa_list`, and in BSD/OS as a normal linked list implemented explicitly. The main reason I mention this change is that it's so simple, everyone is basically doing the same thing, yet three different cases have to be handled. This happens often between the BSD systems (and between them and Linux, too, though not as often) – the same exact thing is done slightly differently by different systems. It would be really helpful if more effort were put into converging such things; this would make it a good bit easier to port code between kernels.

Another common change is that NetBSD and OpenBSD made protocol input or output functions

use varargs rather than fixed parameters (as is still the case in BSD/OS and FreeBSD). This is good in that it allows certain I/O functions to have more parameters added without requiring every input or output function on the entire system to have its argument list adjusted to carry a dummy argument or the type system to be defeated using casts (an unfortunate side effect of the use of a single `struct protosw` for all protocol families). But this is bad in that it introduces a new opportunity for bugs. Arguments expected by the function aren't passed by a caller, so whatever happens to be on the stack becomes the parameter. Also, it is not always so easy to determine when a particular input or output function might not be called (the wonders of function pointers), and using a function pointer to call functions that expect different arguments in the same place creates a bad problem. NetBSD uses the `flags` argument to `ip_output()` to determine which of a few optional arguments are present; this approach might lead to a reasonable solution to the problems I found with using varargs.

There are also post-4.4BSD features that the systems have added, where each system did something different. A great example of this is the systems' choices for addressing TCP SYN flood attacks. BSD/OS implements a SYN cache and leaves the PCBs alone, NetBSD implements a SYN compressed state engine, PCB hash tables, and separate-case PCB lookups, OpenBSD implements SYN cookies and simpler PCB hash tables, and FreeBSD implements PCB hash tables and separate-case PCB lookups. Each change both `tcp_input`'s handling of new connections and the way PCBs lookups are done in significant and different ways, and each requires a special case.

Another interesting change is that NetBSD and FreeBSD pass a `struct proc` around inside the networking stack, from which socket privilege decisions are made, while the old way of doing things as in OpenBSD, BSD/OS is to make that decision when the socket is created and set the `SS_PRIV` flag. The appearance is that this change was made so that sockets lose their privileged status when the process that holds them loses that status; the `proc` that is passed around currently always seems to end up being set to `curproc`. This change has security implications, and was probably made to solve a security problem – but it may create other security problems as side-effects. Neither way appears to be clearly better. But, again, a lot of conditionals get created by this change to carry around this extra argument.

### 3.2 Between BSDs and Linux

Linux is very different than the BSDs. Frequently, either the same thing or a similar thing is done, and these differences aren't so bad to work with.

For example, there are a lot of things between BSD and Linux where almost exactly the same thing has a different name. Linux names its exact-bit types one way – e.g., `__u32` – and BSD names its exact-bit types another way – e.g., `uint32_t`. These types do exactly the same thing. In this case, I have tried to convince the systems' maintainers (with limited success) to make definitions available in kernel space for the POSIX standard names of these types – e.g., `uint32_t` – which would help writers of portable kernel code avoid this particular problem. Another example of this is Linux's `struct iphdr` and BSD's `struct ip`, which have different field names, but both define exactly the same data structure. It would be helpful for portability of system maintainers were to agree either to converge on a single definition for these things or to provide a common definition along with a system-specific definition with another name. Until this happens, these differences can be worked around by using preprocessor `define` statements to do a search and replace.

There are also a lot of things between BSD and Linux that work similarly enough to be interchangeable, especially if you don't need all the functionality that the systems can provide. A great example of this is the difference between Linux's `kmalloc(size, flags)` and BSD's `malloc(size, type, waitflag)`. Both provide a more flexible version of the standard C `malloc(size)` function. By abstracting both into a single `OSDEP_MALLOC(size)` function, they can be used interchangeably on their respective systems.

Another example of this is how the systems handle “fast” critical sections by preventing higher priority interrupt-driven functions from running. Linux uses `save_flags(flags); cli()` and `restore_flags(flags)` for this, which actually turns off CPU interrupts, while BSD uses `s=splnet()` and `splx(s)` to provide this sort of exclusion for network code (other priority levels are used for other types of code), which turns off some software interrupts. For small sections of code which can't delay interrupts long enough to be a problem, these are reasonably equivalent (for longer blocks of code, arguably, neither should be used, and we should really use locks). By abstracting these into `OSDEP_CRITICALDCL`, `OSDEP_CRITICALSTART()`, and `OSDEP_CRITICALEND()`, again, these reasonably equivalent things can be used interchangeably on their respective systems.

Then there are a lot of things that are similar at a high level but not so interchangeable. An example of this is Linux's `sk_buff` and BSD's `mbuf`. We actually took two different approaches to this particular difference, each based on different requirements of different blocks of code. In our PF\_KEY implementation, we needed to form messages in our own data structures and then generate native buffers to pass to the

sockets layer, and we also need to take native buffers passed from the sockets layer and form messages in our own data structures. Also, performance is not critical in this code. So we created interchangeable higher-level functions that copied data into and out of the system-native buffers and performed certain specific operations on those buffers. Examples of these are `OSDEP_DATATOPACKET()`, `OSDEP_ZEROPACKET()`, and `OSDEP_FREEPACKET()`. In our IPsec implementation, we had to work with the buffers without copies, and that required a different approach.

### 3.3 nbufs

One of the really new things that we developed in the course of making our code portable was `struct nbuf`, which is a portable packet buffer structure. We set the design goals of the `nbuf` based on what we considered to be the best features of the Linux `sk_buff`:

- Packet data is contiguous in memory
- Payload data is copied into its final place and the headers are assembled around it in the buffer's "slack space," which helps avoid copies

And we also included what we considered to be the best features of the BSD `mbuf`:

- Small header size
- Few extraneous fields

We also imposed two more requirement, special to how this buffer will be used:

- Converting system-native buffers to `nbufs` must be fast in the common case
- Converting back buffers that were so converted must be fast in the common case

Note that the second requirement is not "converting `nbufs` to system-native buffers must be fast in the common case." This is an important optimization for reduced code complexity that we can make because all of the performance-sensitive paths in our IPsec code take a system-native buffer, convert it to a `nbuf`, work on that, and then convert it back into a system-native buffer. Since we never currently start with a `nbuf` and convert that to a system-native buffer, we don't need to that to be a fast operation. Future uses of the `nbuf` might need that, and we have ideas as to how to do that, but the current implementation does not support that as a fast operation.

The arrangement of the data buffer itself and the contents of the buffer are very similar to that of the Linux `sk_buff`. Both consist of a block of space, with a portion in the middle used for packet data and some

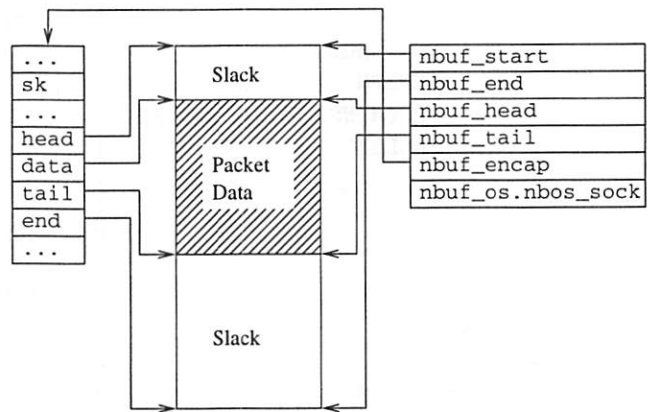


Figure 3: Encapsulation of a `sk_buff` in a `nbuf`

space before and after that left for expansion and/or padding. There are pointers to the start of the buffer (`nbuf_start`), to the end of the buffer (`nbuf_end`), to the head of the packet data (`nbuf_head`), and to the tail of the packet data (`nbuf_tail`).

There are also two special fields used by the border functions. The first, `nbuf_encap`, is a pointer to an "encapsulated" system-native buffer. In border cases that don't involve copies, a `nbuf` header is simply allocated and its fields point into the native buffer, in which case a pointer to the native buffer is kept to make it easy to "convert" back to the native buffer by simply updating the native buffer's fields, freeing the `nbuf` header, and returning the original buffer. The second special field, `nbuf_os`, is a structure that contains system-specific fields that must be copied in the "slow-path" cases where the system-native buffer data is copied to the `nbuf` and the original buffer is destroyed. When the `nbuf` is converted back, most fields in the system-native buffers can be filled in with reasonable default values without problems, but a few non-data fields must be filled in with the "right" original values. The `nbuf_os` structure allows us to ensure that we don't lose those values in the conversions.

Under Linux, conversion from a `sk_buff` to a `nbuf` is a fast path operation so long as enough "slack space" is available for the operations that are about to be performed on the `nbuf`. The Linux network stack already arranges for enough such space to be present in the `sk_buff`, and we extend that to include the overhead of the operations our code performs on the packets, so this fast path conversion should always happen in practice. Figure 3 shows what a `nbuf` looks like on Linux when a `sk_buff` has been encapsulated as part of a fast path conversion. There are some slight differences in the field names, but the fields in each structure are very similar, which makes the mapping between the two very easy.

Size	Typical Values	Typical Arrangement
0 .. MHLEN	0 .. 100	One mbuf
MHLEN+1 .. MCLMINSIZE-1	101 .. 208	Two mbufs or one cluster mbuf
MCLMINSIZE .. MCLBYTES	209 .. 2048	One cluster mbuf
MCLBYTES+1 .. ∞	2049 .. ∞	Multiple cluster mbufs

Figure 2: Typical BSD mbuf Arrangements for Various Sizes

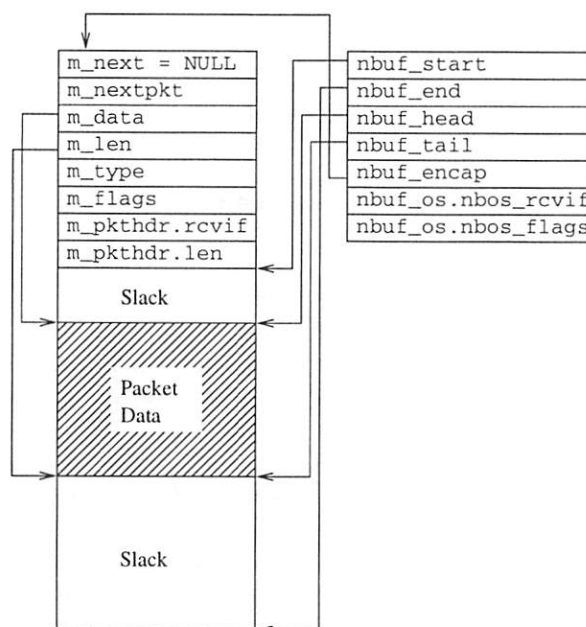


Figure 4: Encapsulation of a normal mbuf in a nbuff

Under BSD, conversion from a mbuf to a nbuff is a fast path operation so long as enough “slack space” is available and as long as all of the packet is contained in one mbuf. In 4.4BSD systems, the arrangement of the packet in buffers typically depends on the packet’s size as shown in Figure 2. Typically seen IP packets tend to be either fairly small (about 44 bytes) or fairly big (about 552, 576, or 1500 bytes) [3]. The key observation we made is that, for the typical path MTU range of 576..1500 bytes, both typically seen categories are contained in one mbuf. Even allowing for the extra packet headers we add on output, we can make a fast path mapping between mbufs and nbuffs for most actually seen packets. This is very important for good common-case performance on BSD systems.

Figure 4 shows how we encapsulate a normal mbuf in a nbuff. The nature of a normal mbuf packet header makes the locations nbuff\_start and nbuff\_end values fixed with respect to the start of the mbuf. The nbuff\_head field is equivalent to the value of m\_data, while the nbuff\_tail field is equivalent to the value of m\_data+m\_len.

Figure 5 shows how we encapsulate a cluster mbuf in a nbuff. The nbuff\_head and nbuff\_tail fields relate to the m\_data and m\_len fields as with a normal mbuf. But now, the values of nbuff\_start and nbuff\_end are not fixed with respect to the mbuf; they are instead equivalent to m\_ext.ext\_buf and m\_ext.ext\_buf+m\_ext.ext\_size, respectively. Note that the data buffer attached to a cluster mbuf is always a size of MCLBYTES on the systems we care about, but we use the value in the field in case that changes.

nbuffs have so far turned out to be very helpful in allowing us to make our IPsec implementation portable without sacrificing common-case performance. We have been looking at other possible uses for them, such as using them on systems different than both BSD and Linux and using them as a graceful transition mechanism between mbufs and a different buffer structure for the BSD network stack.

## 4 Results

### 4.1 Breakdown of Our Tree

Figures 6 and 7, and give some summary statistics about the portability methods we use in our source tree. These should give you a feel for what might be reasonable to expect out of porting kernel code. I interpret these statistics as providing cautious support for the idea of porting code between different kernels.

Figure 7 in particular deserves some extra explanation. Much of the work required for our IPv6 implementation is to “clean up” parts of the the BSD networking stack that were written to be IPv4 only (not an unreasonable assumption, but one that doesn’t hold when we add IPv6). The result is that there are a lot of one line changes, which is why the ratio between changed lines and changed blocks is so close to one. Still, these are all system-specific changes. While the nature of the changes is similar among the different BSD systems, they must be done separately, by hand, and all kept synchronized. This is a lot of work, but the nature of the problem basically requires this approach. We tried to avoid maintaining these separately by using a common netinet tree, but that approach had its own problems. The lesson to be learned here is that there will always be a significant amount of system-

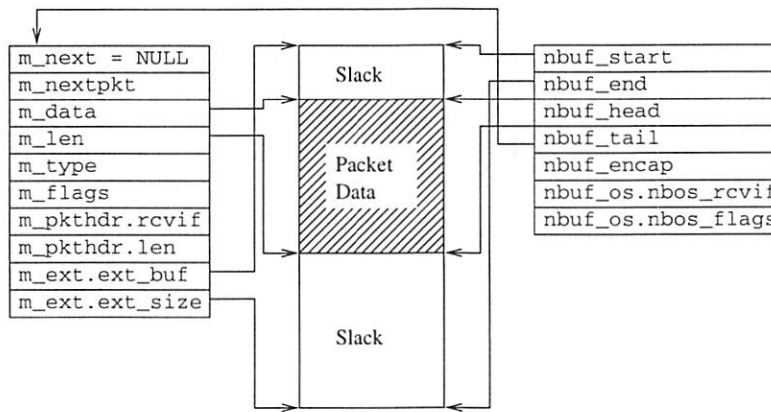


Figure 5: Encapsulation of a cluster mbuf in a nbuf

Category	Blocks	Lines	% Lines
Independent	N/A	14261	45.19
BSD only	N/A	13760	43.60
BSD/OS	28	102	0.32
NetBSD	30	71	0.22
OpenBSD	58	160	0.51
FreeBSD	156	931	2.95
Compound	371	1525	4.83
Linux <sup>1</sup>	44	735	2.33

Figure 6: Conditional code in shared trees

System	Blocks	Lines
BSD/OS <sup>2</sup>	1434	2410
FreeBSD	5388	5575
NetBSD	5080	5247
OpenBSD	4820	4951
Linux <sup>1</sup>	372	563

Figure 7: Changes to the systems' trees

specific code, and certain problems will naturally tend to require more of that.

The good news here is that we have a source tree total of 22270 lines (44.28%) of system-specific code, 13760 lines (27.36%) of BSD-specific code, and 14261 lines (28.36%) of system-independent code. Considering that this counts several copies of effectively the same thing, the nature of the IPv6 changes, and that most of the IPv6 code gets classified as BSD-specific

<sup>1</sup>Our Linux support is a work in progress, for IPsec only, and does not include IPv6. The metrics presented for Linux are for example and aren't directly comparable to the other systems.

<sup>2</sup>Some of our code is already integrated into BSD/OS, which reduces the system-specific changes for that tree.

rather than system-independent, this is pretty good. More than a quarter of our code is portable, and more than a quarter of our code at least runs on all of the BSDs we support.

The source tree totals penalize system-specific code more heavily when more systems are considered; if I simply omitted all support for a system, the percentage of system-independent and BSD-specific code would go up, and those percentages would give the illusion that more things are portable, even though removal of support for a system means the opposite is really true. Another way to look at these results that doesn't have this problem is to consider the breakdown of the code that goes into actual systems. Figure 8 shows the results of such a breakdown. On the BSD systems, the portable components make up about 40% each of the lines of code, with the system-specific components only being about 20%. On Linux, where we only support IPsec, the portable components make up 83.48% of the lines and the system-specific components make up 16.52%.

## 4.2 Conclusions

Based on these statistics, I conclude that, for code that can be reasonably ported, about one to two fifths of the source code will need to be written as system-specific code for each port and about three to four fifths of the source code can be made portable. That's still much better than half in common, and I believe that this provides cautious support for the idea that porting kernel code between significantly different systems can be a very practical and worthwhile thing to do.

Certain things are going to be inherently more or less system-specific. In our case, we had one thing that was not inherently very system specific (IPsec) and one thing that was more inherently system specific (IPv6). Even with the latter, we still achieved a good amount of portability. This is promising in that it suggests

System	System-specific	Lines (%) of Code	
		BSD-specific	System-independent
BSD/OS <sup>2</sup>	4037 (12.59)	13760 (42.92)	14261 (44.48)
FreeBSD	8031 (22.28)	13760 (38.17)	14261 (39.56)
NetBSD	6843 (19.63)	13760 (39.47)	14261 (40.90)
OpenBSD	6505 (18.84)	13760 (39.85)	14261 (41.31)
Linux <sup>1</sup>	2823 (16.52)	N/A	14261 (83.48)

Figure 8: Breakdown of Code by System<sup>3</sup>

that a broad class of things could be practical to make portable.

The five systems that we support are different, yet they are still very similar. Proponents of some of these systems will try hard to deny this, especially in the case of the differences between BSD and Linux, but code doesn't lie. At least, not as much.

## 5 Acknowledgments

The NRL IPv6+IPsec distribution is the result of years of work from a lot of people. Other than myself, the implementation team includes or has included Randall Atkinson, Ken Chin, Daniel McDonald, Ronald Lee, Bao Phan, Chris Telfer, and Chris Winters. The software's evolution and a lot of our portability efforts were the combined result of everyone's effort, and they are each at least as deserving of credit as I am.

The most recent portability work, including our current source tree organization and `nbufs`, was done by myself, Ronald Lee, Chris Telfer, and Chris Winters.

I'd like to thank those members of the communities surrounding the systems we support who have helped us. In particular, David Borman, Alan Cox, Theo de Raadt, and Perry Metzger. If it weren't for their help, we wouldn't be able to currently support the systems we do.

And, of course, we all must not forget to thank the developers of the systems that we run on, especially for the free systems. It's a *lot* of work to develop and maintain an entire operating system. It's amazing that small groups of people can do this, frequently in their "spare time," and produce such high-quality results. If they didn't do this and make the source so easily available (if not free), my group at NRL, and many more like us, might not have systems to develop on, or we might not be able to give our results away freely.

Ronald Lee and Angelos Keromytis provided helpful feedback on earlier drafts of this paper.

<sup>3</sup> "Complex" system-specific blocks are counted as if they were included as system-specific blocks in all systems. This slightly under-represents the portable components.

The work described in this paper was done at the Center for High Assurance Computer Systems at the U.S. Naval Research Laboratory. This work was sponsored by the Information Technology Office, Defense Advanced Research Projects Agency (DARPA/ITO) as part of our Internet Security Technology project and by the Security Program Office (PMW-161), U.S. Space and Naval Warfare Systems Command (SPAWAR). I and my co-workers really appreciate their sponsorship of NRL's network security efforts and their continued support of IPsec development. Without that support, this paper, and this software, would not exist.

## References

- [1] Randall J. Atkinson, Ken E. Chin, Bao G. Phan, Daniel L. McDonald, and Craig Metz. Implementation of IPv6 in 4.4BSD. *Proceedings of the 1996 Usenix Annual Technical Conference*, January 1996.
- [2] D. L. McDonald, C. W. Metz, and B. G. Phan. PF\_KEY Key Management API, Version 2, RFC 2367, July 1998.
- [3] K. Claffy, Greg Miller, and Kevin Thompson. The Nature of the Beast: Recent Traffic Measurements From an Internet Backbone. *Proceedings of INET '98*, July 1998.

In addition to formal references, this work and this paper refers heavily to the source code for the five systems. For more information on each, go to:

BSD/OS	<a href="http://www.bsdi.com">http://www.bsdi.com</a>
FreeBSD	<a href="http://www.freebsd.org">http://www.freebsd.org</a>
NetBSD	<a href="http://www.netbsd.org">http://www.netbsd.org</a>
OpenBSD	<a href="http://www.openbsd.org">http://www.openbsd.org</a>
Linux	<a href="http://www.linux.org">http://www.linux.org</a>

For more information about the NRL IPv6+IPsec distribution, or to obtain the code, go to:  
<http://www.ipv6.nrl.navy.mil>

# strncpy and strlcat — consistent, safe, string copy and concatenation.

*Todd C. Miller*  
*University of Colorado, Boulder*  
*Theo de Raadt*  
*OpenBSD project*

## Abstract

As the prevalence of buffer overflow attacks has increased, more and more programmers are using size or length-bounded string functions such as `strncpy()` and `strncat()`. While this is certainly an encouraging trend, the standard C string functions generally used were not really designed for the task. This paper describes an alternate, intuitive, and consistent API designed with safe string copies in mind.

There are several problems encountered when `strncpy()` and `strncat()` are used as safe versions of `strcpy()` and `strcat()`. Both functions deal with NUL-termination and the length parameter in different and non-intuitive ways that confuse even experienced programmers. They also provide no easy way to detect when truncation occurs. Finally, `strncpy()` zero-fills the remainder of the destination string, incurring a performance penalty. Of all these issues, the confusion caused by the length parameters and the related issue of NUL-termination are most important. When we audited the OpenBSD source tree for potential security holes we found rampant misuse of `strncpy()` and `strncat()`. While not all of these resulted in exploitable security holes, they made it clear that the rules for using `strncpy()` and `strncat()` in safe string operations are widely misunderstood. The proposed replacement functions, `strlcpy()` and `strlcat()`, address these problems by presenting an API designed for safe string copies (see Figure 1 for function prototypes). Both functions guarantee NUL-termination, take as a length parameter the size of the string in bytes, and provide an easy way to detect truncation. Neither function zero-fills unused bytes in the destination.

## Introduction

In the middle of 1996, the authors, along with other members of the OpenBSD project, undertook an audit of the OpenBSD source tree looking for security problems, starting with an emphasis on buffer overflows. Buffer overflows [1] had recently gotten a lot of attention in forums such as BugTraq [2] and were being widely exploited. We found a large number of overflows due to unbounded string copies using `sprintf()`, `strcpy()` and `strcat()`, as well as loops that manipulated strings without an explicate length check in the loop invariant. Additionally, we also found many instances where the programmer had tried to do safe string manipulation with `strncpy()` and `strncat()` but failed to grasp the subtleties of the API.

Thus, when auditing code, we found that not only was it necessary to check for unsafe usage of functions like `strcpy()` and `strcat()`, we also had to check for incorrect usage of `strncpy()` and `strncat()`. Checking for correct usage is not always obvious, especially in the case of “static” variables or buffers allocated via `calloc()`, which are effectively pre-terminated. We came to the conclusion that a foolproof alternative to `strncpy()` and `strncat()` was needed, primarily to simplify the job

of the programmer, but also to make code auditing easier.

---

```
size_t strlcpy(char *dst, \
               const char *src, size_t size);
size_t strlcat(char *dst, \
               const char *src, size_t size);
```

**Figure 1:** ANSI C prototypes for `strlcpy()` and `strlcat()`

---

## Common Misconceptions

The most common misconception is that `strncpy()` NUL-terminates the destination string. This is only true, however, if length of the source string is less than the size parameter. This can be problematic when copying user input that may be of arbitrary length into a fixed size buffer. The safest way to use `strncpy()` in this situation is to pass it one less than the size of the destination string, and then terminate the string by hand. That way you are guaranteed to always have a NUL-terminated destination string. Strictly speaking, it

is not necessary to hand-terminate the string if it is a "static" variable or if it was allocated via `calloc()` since such strings are zeroed out when allocated. However, relying on this feature is generally confusing to those persons who must later maintain the code.

There is also an implicate assumption that converting code from `strcpy()` and `strcat()` to `strncpy()` and `strncat()` causes negligible performance degradation. With this is true of `strncat()`, the same cannot be said for `strncpy()` since it zero-fills the remaining bytes not used to store the string being copied. This can lead to a measurable performance hit when the size of the destination string is much greater than the length of the source string. The exact penalty for using `strncpy()` due to this behavior varies by CPU architecture and implementation.

The most common mistake made with `strncat()` is to use an incorrect size parameter. While `strncat()` does guarantee to NUL-terminate the destination, you must not count the space for the NUL in the size parameter. Most importantly, this is not the size of the destination string itself, rather it is the amount of space available. As this is almost always a value that must be computed, as opposed to a known constant, it is often computed incorrectly.

## How do `strncpy()` and `strlcat()` help things?

The `strncpy()` and `strlcat()` functions provide a consistent, unambiguous API to help the programmer write more bullet-proof code. First and foremost, both `strncpy()` and `strlcat()` guarantee to NUL-terminate the destination string for all strings where the given size is non-zero. Secondly, both functions take the full size of the destination string as a size parameter. In most cases this value is easily computed at compile time using the `sizeof` operator. Finally, neither `strncpy()` nor `strlcat()` zero-fill their destination strings (other than the compulsory NUL to terminate the string).

The `strncpy()` and `strlcat()` functions return the total length of the string they tried to create. For `strncpy()` that is simply the length of the source; for `strlcat()` that means the length of the destination (before concatenation) plus the length of the source. To check for truncation, the programmer need only verify that the return value is less than the size parameter. Thus, if truncation has occurred, the number of bytes needed to store the entire string is now known and the programmer may allocate more space and re-copy the strings if he or she wishes. The return value has similar semantics to the return value of `snprintf()` as implemented in BSD and as specified by the upcoming C9X specification [4] (note that not all `snprintf()` implementations

currently comply with C9X). If no truncation occurred, the programmer now has the length of the resulting string. This is useful since it is common practice to build a string with `strncpy()` and `strncat()` and then to find the length of the result using `strlen()`. With `strncpy()` and `strlcat()` the final `strlen()` is no longer necessary.

Example 1a is a code fragment with a potential buffer overflow (the `HOME` environment variable is controlled by the user and can be of arbitrary length).

---

```
strcpy(path, homedir);
strcat(path, "/");
strcat(path, ".foorc");
len = strlen(path);
```

**Example 1a:** Code fragment using `strcpy()` and `strcat()`

---

Example 1b is the same fragment converted to safely use `strncpy()` and `strncat()` (note that we have to terminate the destination by hand).

---

```
strncpy(path, homedir,
        sizeof(path) - 1);
path[sizeof(path) - 1] = '\0';
strncat(path, "/",
        sizeof(path) - strlen(path) - 1);
strncat(path, ".foorc",
        sizeof(path) - strlen(path) - 1);
len = strlen(path);
```

**Example 1b:** Converted to `strncpy()` and `strncat()`

---

Example 1c is a trivial conversion to the `strncpy()/strlcat()` API. It has the advantage of being as simple as Example 1a, but it does not take advantage of the new API's return value.

---

```
strncpy(path, homedir, sizeof(path));
strlcat(path, "/", sizeof(path));
strlcat(path, ".foorc", sizeof(path));
len = strlen(path);
```

**Example 1c:** Trivial conversion to `strncpy()/strlcat()`

---

Since Example 1c is so easy to read and comprehend, it

is simple to add additional checks to it. In Example 1d, we check the return value to make sure there was enough space for the source string. If there was not, we return an error. This is slightly more complicated but in addition to being more robust, it also avoids the final `strlen()` call.

---

```
len = strncpy(path, homedir,
              sizeof(path));
if (len >= sizeof(path))
    return (ENAMETOOLONG);
len = strlcat(path, "/",
             sizeof(path));
if (len >= sizeof(path))
    return (ENAMETOOLONG);
len = strlcat(path, ".foorc",
             sizeof(path));
if (len >= sizeof(path))
    return (ENAMETOOLONG);
```

**Example 1d:** Now with a check for truncation

---

### Design decisions

A great deal of thought (and a few strong words) went into deciding just what the semantics of `strncpy()` and `strlcat()` would be. The original idea was to make `strncpy()` and `strlcat()` identical to `strncpy()` and `strncat()` with the exception that they would always NUL-terminate the destination string. However, looking back on the common use (and misuse) of `strncat()` convinced us that the size parameter for `strlcat()` should be the full size of the string and not just the number of characters left unallocated. The return values started out as the number of characters copied, since this was trivial to get as a side effect of the copy or concatenation. We soon decided that a return value with the same semantics as `snprintf()`'s was a better choice since it gives the programmer the most flexibility with respect to truncation detection and recovery.

### Performance

Programmers are starting to avoid `strncpy()` due its poor performance when the target buffer is significantly larger than the length of the source string. For instance, the apache group [6] replaced calls to `strncpy()` with an internal function and noticed a performance improvement [7]. Also, the ncurses [8] package recently removed an occurrence of `strncpy()`, resulting in a factor of four speedup of the *tic* utility. It is our hope that, in the future, more programmers will use the

interface provided by `strlcpy()` rather than using a custom interface.

To get a feel for the worst-case scenario in comparing `strncpy()` and `strlcpy()`, we ran a test program that copies the string "this is just a test" 1000 times into a 1024 byte buffer. This is somewhat unfair to `strncpy()`, since by using a small string and a large buffer `strncpy()` has to fill most of the buffer with NUL characters. In practice, however, it is common to use a buffer that is much larger than the expected user input. For instance, pathname buffers are `MAXPATHLEN` long (1024 bytes), but most filenames are significantly shorter than that. The averages run times in Table 1 were generated on an HP9000/425t with a 25Mhz 68040 CPU running OpenBSD 2.5 and a DEC AXP-PCI166 with a 166Mhz alpha CPU also running OpenBSD 2.5. In all cases, the same C versions of the functions were used and the times are the "real time" as reported by the *time* utility.

cpu	function	time
m68k	strcpy	0.137
m68k	strncpy	0.464
m68k	strlcpy	0.14
alpha	strcpy	0.018
alpha	strncpy	0.10
alpha	strlcpy	0.02

**Table 1:** Performance timings in seconds

As can be seen in Table 1, the timings for `strncpy()` are far worse than those for `strcpy()` and `strlcpy()`. This is probably due not only to the cost of NUL padding but also because the CPU's data cache is effectively being flushed by the long stream of zeroes.

### What `strlcpy()` and `strlcat()` are not

While `strlcpy()` and `strlcat()` are well-suited for dealing with fixed-size buffers, they cannot replace `strncpy()` and `strncat()` in all cases. There are still times where it is necessary to manipulate buffers that are not true C strings (the strings in `struct utmp` for instance). However, we would argue that such "pseudo strings" should not be used in new code since they are prone to misuse, and in our experience, a common source of bugs. Additionally, the `strlcpy()` and `strlcat()` functions are not an attempt to "fix" string handling in C, they are designed to fit within the normal framework of C strings. If you require string functions that support dynamically allocated, arbitrary sized buffers you may wish to examine the "astring" package from mib software [9].

## Who uses `strncpy()` and `strncat()`?

The `strncpy()` and `strncat()` functions first appeared in OpenBSD 2.4. The functions have also recently been approved for inclusion in a future version of Solaris. Third-party packages are starting to pick up the API as well. For instance, the `rsync` [5] package now uses `strncpy()` and provides its own version if the OS does not support it. It is our hope that other operating systems and applications will use `strncpy()` and `strncat()` in the future, and that it will receive standards acceptance at some time.

## What's Next?

We plan to replace occurrences of `strncpy()` and `strncat()` with `strncpy()` and `strncat()` in OpenBSD where it is sensible to do so. While new code in OpenBSD is being written to use the new API, there is still a large amount of code that was converted to use `strncpy()` and `strncat()` during our original security audit. To this day, we continue to discover bugs due to incorrect usage of `strncpy()` and `strncat()` in existing code. Updating older code to use `strncpy()` and `strncat()` should serve to speed up some programs and uncover bugs in others.

## Availability

The source code for `strncpy()` and `strncat()` is available free of charge and under a BSD-style license as part of the OpenBSD operating system. You may also download the code and its associated manual pages via anonymous ftp from `ftp.openbsd.org` in the directory `/pub/OpenBSD/src/lib/libc/string`. The source code for `strncpy()` and `strncat()` is in `strncpy.c` and `strncat.c`. The documentation (which uses the `tmac.doc` troff macros) may be found in `strncpy.3`.

## Author Information

Todd C. Miller has been involved in the free software community since 1993 when he took over maintenance of the `sudo` package. He joined the OpenBSD project in 1996 as an active developer. Todd belatedly received a BS in Computer Science in 1997 from the University of Colorado, Boulder (after years of prodding). Todd has so far managed to avoid the corporate world and currently works as a Systems Administrator at the University of Colorado, Boulder blissfully ensconced in academia. He may be reached via email at `<Todd.Miller@cs.colorado.edu>`.

Theo de Raadt has been involved with free Unix operating systems since 1990. Early developments included porting Minix to the `sun3/50` and `amiga`, and

PDP-11 BSD 2.9 to a 68030 computer. As one of the founders of the NetBSD project, Theo worked on maintaining and improving many system components including the `sparc` port and a free YP implementation that is now in use by most free systems. In 1995 Theo created the OpenBSD project, which places focus on security, integrated cryptography, and code correctness. Theo works full time on advancing OpenBSD. He may be reached via email at `<deraadt@openbsd.org>`.

## References

- [1] Aleph One. "Smashing The Stack For Fun And Profit." *Phrack Magazine Volume Seven, Issue Forty-Nine*.
- [2] BugTraq Mailing List Archives. <http://www.geek-girl.com/bugtraq/>. This web page contains searchable archives of the BugTraq mailing list.
- [3] Brian W. Kernighan, Dennis M. Ritchie. *The C Programming Language, Second Edition*. Prentice Hall, PTR, 1988.
- [4] International Standards Organization. "C9X FCD, Programming languages — C" <http://wwwold.dkuug.dk/jtc1/sc22/open/n2794/>. This web page contains the current draft of the upcoming C9X standard.
- [5] Andrew Tridgell, Paul Mackerras. *The rsync algorithm*. [http://rsync.samba.org/rsync/tech\\_report/](http://rsync.samba.org/rsync/tech_report/). This web page contains a technical report describing the `rsync` program.
- [6] The Apache Group. The Apache Web Server. <http://www.apache.org>. This web page contains information on the Apache web server.
- [7] The Apache Group. New features in Apache version 1.3. [http://www.apache.org/docs/new\\_features\\_1\\_3.html](http://www.apache.org/docs/new_features_1_3.html). This web page contains new features in version 1.3 of the Apache web server.
- [8] The Ncurses (new curses) home page. <http://www.clark.net/pub/dickey/ncurses/>. This web page contains Ncurses information and distributions.
- [9] Forrest J. Cavalier III. "Libmib allocated string functions." <http://www.mibsoftware.com/libmib/astring/>. This web page contains a description and implementation of a set of string functions that dynamically allocate memory as necessary.

# *pk*: A POSIX Threads Kernel

Frank W. Miller  
Cornfed Systems, Inc.  
[www.cornfed.com](http://www.cornfed.com)

## Introduction

*pk* is a new operating system kernel targeted for use in real-time and embedded applications. There are two novel aspects to the *pk* design:

- *Documentation*: The kernel is documented using *literate programming* techniques and the *noweb* [4] tool in particular.
- *POSIX Threads with Memory Protection*: The concurrency model is based on the POSIX Threads (aka Pthreads [2, 3]) standard, however, the kernel also provides page-based memory protection using Memory Management Unit (MMU) hardware.

This short paper discusses these facets of the *pk* kernel project. The use of literate programming is presented first, followed by a brief description of some of the *pk* design issues.

## Literate Programming

Documentation is as important as the software it documents. This belief led me to contemplate how to document the *pk* kernel design as a primary goal. In my experience, the biggest problem with generating documentation is that it often seems to be a secondary activity, performed after the code is written. I became interested in the potential for the *documentation discipline* associated with literate programming techniques and decided to make use of these techniques with *pk*.

By discipline, I refer to a structure within which a project is performed that provides an incentive to generate documentation as the code is being written. Literate programming tools provide a mechanism that fosters such structure.

*pk* makes use of a literate programming tool called *noweb*. The basic concept is simple. Both documentation and code are contained in a single *noweb* file that uses several special formatting conventions. Two tools are provided. *noweave* extracts the documentation portion of the *noweb* file and generates a documentation file, in this case a  $\text{\LaTeX}$  file. *notangle* extracts the source code portion of the *noweb* file and generates a source code file, in this case C source code.

The main consequence of using literate programming, and *noweb* in this case, is that changes to the system after initial development are performed on the *noweb* source files. Since the code is intermixed with its associated documentation, it is more likely that the documentation will be updated at the same time.

This is the first non-trivial project I have undertaken using literate programming, and I have seen an evolution in my use of the tools as I have progressed with it. As with many projects, it was begun drawing on code from another project. In this case, I drew on some elements of the *Roadrunner* operating system [1]. These were basic elements, like initialization, interrupt processing, and memory management, that were needed to get a new kernel up and running quickly. These reused elements were not documented with *noweb* initially and some remain undocumented still although my goal is to document the entire system using *noweb* over time.

The first new element to be written was the set of basic Pthread routines. I first wrote the code and only after it was completed and tested to some degree, did I go back and overlay the documentation and formatting to turn the C source code files into *noweb* files. This pattern repeated itself during the implementation of mutexes and condition variables. *noweb* documentation was added only after the fact.

It happened that once I had completed the Pthreads routines, I decided to investigate the addition of pro-

tected memory to the kernel. Design issues associated with this decision are discussed in the next section. Continuing here, I want to discuss the implications on documentation that presented themselves. It was this decision that resulted in the first significant changes to existing source code that had been documented using *noweb*. Specifically, the memory management code, which maintains the heap of available physical pages, and various parts of the Pthreads code needed to be updated.

My first thought when I went to make changes to the first source code file was, "don't worry about the documentation, you can come back a fix that up later." I had no sooner opened my second source file when I realized I would forget what I had done if I didn't take care of the documentation. This would result in a document whose prose didn't match the code associated with it. I *had* to go back and change it. This was the discipline I had hoped would present itself. I went back and made the documentation changes.

At first this felt cumbersome, it added time to code maintenance. However, two unexpected effects began to emerge. First, I found that my design was cleaner. When I modified the code *and* changed the documentation, I thought about the problem twice. This led in several instances to a more concise change. Second, I found that I could make changes more quickly in code that I had not visited in a while. It may seem obvious, but the documentation was right there next to the code, and this allowed me to refamiliarize myself with it more quickly. I have now begun to implement pieces of code in the *noweb* source format as they are written for the first time. The power of the conciseness effect I discovered during maintenance is also present when writing code and documentation together during an initial implementation.

The granularity of the documentation varies over different parts of the code. There are several reasons for this. Foremost, different areas of the code have been documented at different times, and the documentation for a particular segment of code might not be performed all at one sitting. This results in sections of code that are "complete", i.e. they are documented in great enough detail to understand all aspects of their semantics. Other portions are coarser, perhaps only setup to fit into the overall structure of the piece of documentation, but not yet completed. There are also portions of code that do not require heavy documentation. They may be

small routines or the code itself may be so intuitive that only high-level prose is required to get across their function.

One interesting point about the use of literate programming seems to be that the licensing associated with the source code must also apply to the documentation, since the two are linked in the source files. *pk* is available under a BSD-style copyright, which places essentially no restrictions on redistribution. A similar project released under the Gnu Public License (GPL) would require the changes to the documentation to be redistributed in addition to changes source code.

There are a variety of literate programming tools available. I evaluated *cweb*, written by Donald Knuth, and *noweb* written by Norman Ramsey. Knuth's *cweb* generates documentation of code fragments that are "pretty-printed", i.e. they have an algorithmic style reminiscent of textbooks on computer science theory. The *noweb* tools utilize a small set of simple formatting rules and generate code fragments that look cosmetically like they were extracted from a source code file. This style seemed more in tune with a systems programming project, like an operating system kernel, and so I decided to use *noweb* over *cweb*.

## Pthreads and Memory Protection

*pk* is based on the POSIX Threads concurrency model. Pthreads were originally designed under the assumption that all of the threads would execute in the same address space. In fact, this address space was intended to be within a UNIX process. However, the Pthreads API is also used in real-time kernels that provide their applications a single, physical address space. *pk* is also targeted at real-time and embedded applications, but it augments the Pthreads design to include page-based memory protection using the MMU. Such a design falls somewhere in between the basic Pthreads model and the more substantial process concurrency model.

Since *pk* is targeted at time-critical applications, paging and/or swapping to secondary storage cannot be utilized. This is because of the significant lack of determinism introduced by moving memory pages back and forth to secondary storage.

If neither paging or swapping is used, applications are limited to the amount of physical memory on a given machine. This fact raised the question of whether providing separate address spaces for each thread, in a manner akin to a process, was desirable. In my experience in embedded systems development, having direct access to specific physical addresses can be useful. For this reason, I decided to map virtual to physical addresses one-to-one.

The MMU is used simply to restrict access to memory locations, not to provide separate address spaces. Three types of memory protection are provided:

**Inter-thread:** Threads may not access memory belonging to another thread.

**Kernel-thread:** Threads may not access kernel memory except through well-defined system call entry points.

**Intra-thread:** Code segments associated with a thread can be marked read-only.

Restricting access to parts of memory violates the assumption of a single unprotected address space present in the Pthreads API design. There are a variety of parameters in the API where pointers capable of referencing arbitrary memory locations are utilized. Allowing arbitrary values to be passed through these parameters invites the generation of copious page and general protection faults.

Several areas of the API have been scrutinized to address potential problems. The following list illustrates some of the attention required for the Pthreads API in *pk*. The list is not exhaustive, but gives a flavor of the kinds of issues in the API that cause difficulty in the *pk* design.

**Data Structures:** Several data types have been further specified. Reference types for pthreads (`pthread_t`), mutexes (`pthread_mutex_t`), and condition variables (`pthread_cond_t`) cannot be typed as arbitrary pointers. In *pk*, they are defined as integer indices into kernel tables.

**pthread\_create():** The values for the `start` function pointer and `arg` argument pointer represent potentially arbitrary pointer accesses. In *pk*, semantic restrictions are placed on these

pointer values. They must each point to the beginning of a valid region and the ownership and mappings for each region will be transferred to the new thread if the creation succeeds.

**pthread\_exit()** The `retval` return value can be an arbitrary pointer value. The return value type is changed to `int` in *pk*.

**pthread\_join()** The `retval` parameter is used to collect the return value from an exiting thread. This parameters type is changed to `int` in *pk*.

Several of these changes represent further specification of parts of the Pthreads standard that are not explicit. Changing the type of the return value represents a deviation from the standard. It is hoped that the impact of this change is minimal in code that might be ported to the *pk* system.

## Conclusion

*pk* is available under BSD-style copyright terms. More information on the kernel is available on the web at [www.cornfed.com/pk](http://www.cornfed.com/pk). Downloads of source code and bootable floppy disk images are available at [ftp.cornfed.com/pub](http://ftp.cornfed.com/pub). At the time of this short paper submission, late April 1999, there have been approximately 650 downloads of the *pk* source code distribution in the four months since its initial release was December 21, 1998. The interest in the system is quite gratifying and I look forward to continued and expanded development.

## References

- [1] Cornfed Systems, Inc., *Roadrunner Operating System Reference*, [www.cornfed.com](http://www.cornfed.com).
- [2] IEEE, *POSIX Std 1003.1c*, [www.ieee.org](http://www.ieee.org).
- [3] Pthreads, [www.mit.edu/people/proven/pthreads.html](http://www.mit.edu/people/proven/pthreads.html).
- [4] Ramsey, N., *Noweb — A Simple, Extensible Tool for Literate Programming*, <http://www.cs.virginia.edu/nr/noweb>.



# Berkeley DB

Michael A. Olson

Keith Bostic

Margo Seltzer

*Sleepycat Software, Inc.*

## Abstract

Berkeley DB is an Open Source embedded database system with a number of key advantages over comparable systems. It is simple to use, supports concurrent access by multiple users, and provides industrial-strength transaction support, including surviving system and disk crashes. This paper describes the design and technical features of Berkeley DB, the distribution, and its license.

## 1. Introduction

The Berkeley Database (Berkeley DB) is an embedded database system that can be used in applications requiring high-performance concurrent storage and retrieval of key/value pairs. The software is distributed as a library that can be linked directly into an application. It provides a variety of programmatic interfaces, including callable APIs for C, C++, Perl, Tcl and Java. Users may download Berkeley DB from Sleepycat Software's Web site, at [www.sleepycat.com](http://www.sleepycat.com).

Sleepycat distributes Berkeley DB as an Open Source product. The company collects license fees for certain uses of the software and sells support and services.

### 1.1. History

Berkeley DB began as a new implementation of a hash access method to replace both `hsearch` and the various `dbm` implementations (`dbm` from AT&T, `ndbm` from Berkeley, and `gdbm` from the GNU project). In 1990 Seltzer and Yigit produced a package called Hash to do this [Selt91].

The first general release of Berkeley DB, in 1991, included some interface changes and a new B+tree access method. At roughly the same time, Seltzer and Olson developed a prototype transaction system based on Berkeley DB, called LIBTP [Selt92], but never released the code.

The 4.4BSD UNIX release included Berkeley DB 1.85 in 1992. Seltzer and Bostic maintained the code in the early 1990s in Berkeley and in Massachusetts. Many users adopted the code during this period.

By mid-1996, users wanted commercial support for the software. In response, Bostic and Seltzer formed Sleepycat Software. The company enhances,

distributes, and supports Berkeley DB and supporting software and documentation. Sleepycat released version 2.1 of Berkeley DB in mid-1997 with important new features, including support for concurrent access to databases. The company makes about three commercial releases a year, and most recently shipped version 2.8.

### 1.2. Overview of Berkeley DB

The C interfaces in Berkeley DB permit `dbm`-style record management for databases, with significant extensions to handle duplicate data items elegantly, to deal with concurrent access, and to provide transactional support so that multiple changes can be simultaneously committed (so that they are made permanent) or rolled back (so that the database is restored to its state at the beginning of the transaction).

C++ and Java interfaces provide a small set of classes for operating on a database. The main class in both cases is called `Db`, and provides methods that encapsulate the `dbm`-style interfaces that the C interfaces provide.

Tcl and Perl interfaces allow developers working in those languages to use Berkeley DB in their applications. Bindings for both languages are included in the distribution.

Developers may compile their applications and link in Berkeley DB statically or dynamically.

### 1.3. How Berkeley DB is used

The Berkeley DB library supports concurrent access to databases. It can be linked into standalone applications, into a collection of cooperating applications, or into servers that handle requests and do database

operations on behalf of clients.

Compared to using a standalone database management system, Berkeley DB is easy to understand and simple to use. The software stores and retrieves records, which consist of key/value pairs. Keys are used to locate items and can be any data type or structure supported by the programming language.

The programmer can provide the functions that Berkeley DB uses to operate on keys. For example, B+trees can use a custom comparison function, and the Hash access method can use a custom hash function. Berkeley DB uses default functions if none are supplied. Otherwise, Berkeley DB does not examine or interpret either keys or values in any way. Values may be arbitrarily long.

It is also important to understand what Berkeley DB is not. It is not a database server that handles network requests. It is not an SQL engine that executes queries. It is not a relational or object-oriented database management system.

It is possible to build any of those on top of Berkeley DB, but the package, as distributed, is an embedded database engine. It has been designed to be portable, small, fast, and reliable.

#### 1.4. Applications that use Berkeley DB

Berkeley DB is embedded in a variety of proprietary and Open Source software packages. This section highlights a few of the products that use it.

Directory servers, which do data storage and retrieval using the Local Directory Access Protocol (LDAP), provide naming and directory lookup service on local-area networks. This service is, essentially, database query and update, but uses a simple protocol rather than SQL or ODBC. Berkeley DB is the embedded data manager in the majority of deployed directory servers today, including LDAP servers from Netscape, MessageDirect (formerly Isode), and others.

Berkeley DB is also embedded in a large number of mail servers. Intermail, from Software.com, uses Berkeley DB as a message store and as the backing store for its directory server. The sendmail server (including both the commercial Sendmail Pro offering from Sendmail, Inc. and the version distributed by sendmail.org) uses Berkeley DB to store aliases and other information. Similarly, Postfix (formerly VMailer) uses Berkeley DB to store administrative information.

In addition, Berkeley DB is embedded in a wide variety of other software products. Example applications

include managing access control lists, storing user keys in a public-key infrastructure, recording machine-to-network-address mappings in address servers, and storing configuration and device information in video post-production software.

Finally, Berkeley DB is a part of many other Open Source software packages available on the Internet. For example, the software is embedded in the Apache Web server and the Gnome desktop.

## 2. Access Methods

In database terminology, an access method is the disk-based structure used to store data and the operations available on that structure. For example, many database systems support a B+tree access method. B+trees allow equality-based lookups (find keys equal to some constant), range-based lookups (find keys between two constants) and record insertion and deletion.

Berkeley DB supports three access methods: B+tree, Extended Linear Hashing (Hash), and Fixed- or Variable-length Records (Recno). All three operate on records composed of a key and a data value. In the B+tree and Hash access methods, keys can have arbitrary structure. In the Recno access method, each record is assigned a record number, which serves as the key. In all the access methods, the value can have arbitrary structure. The programmer can supply comparison or hashing functions for keys, and Berkeley DB stores and retrieves values without interpreting them.

All of the access methods use the host filesystem as a backing store.

### 2.1. Hash

Berkeley DB includes a Hash access method that implements extended linear hashing [Litw80]. Extended linear hashing adjusts the hash function as the hash table grows, attempting to keep all buckets underfull in the steady state.

The Hash access method supports insertion and deletion of records and lookup by exact match only. Applications may iterate over all records stored in a table, but the order in which they are returned is undefined.

### 2.2. B+tree

Berkeley DB includes a B+tree [Come79] access method. B+trees store records of key/value pairs in leaf pages, and pairs of (key, child page address) at internal nodes. Keys in the tree are stored in sorted order,

where the order is determined by the comparison function supplied when the database was created. Pages at the leaf level of the tree include pointers to their neighbors to simplify traversal. B+trees support lookup by exact match (equality) or range (greater than or equal to a key). Like Hash tables, B+trees support record insertion, deletion, and iteration over all records in the tree.

As records are inserted and pages in the B+tree fill up, they are split, with about half the keys going into a new peer page at the same level in the tree. Most B+tree implementations leave both nodes half-full after a split. This leads to poor performance in a common case, where the caller inserts keys in order. To handle this case, Berkeley DB keeps track of the insertion order, and splits pages unevenly to keep pages fuller. This reduces tree size, yielding better search performance and smaller databases.

On deletion, empty pages are coalesced by reverse splits into single pages. The access method does no other page balancing on insertion or deletion. Keys are not moved among pages at every update to keep the tree well-balanced. While this could improve search times in some cases, the additional code complexity leads to slower updates and is prone to deadlocks.

For simplicity, Berkeley DB B+trees do no prefix compression of keys at internal or leaf nodes.

### 2.3. Recno

Berkeley DB includes a fixed- or variable-length record access method, called *Recno*. The *Recno* access method assigns logical record numbers to each record, and can search for and update records by record number. *Recno* is able, for example, to load a text file into a database, treating each line as a record. This permits fast searches by line number for applications like text editors [Ston82].

*Recno* is actually built on top of the B+tree access method and provides a simple interface for storing sequentially-ordered data values. The *Recno* access method generates keys internally. The programmer's view of the values is that they are numbered sequentially from one. Developers can choose to have records automatically renumbered when lower-numbered records are added or deleted. In this case, new keys can be inserted between existing keys.

## 3. Features

This section describes important features of Berkeley DB. In general, developers can choose which features are useful to them, and use only those that are required

by their application.

For example, when an application opens a database, it can declare the degree of concurrency and recovery that it requires. Simple stand-alone applications, and in particular ports of applications that used *dbm* or one of its variants, generally do not require concurrent access or crash recovery. Other applications, such as enterprise-class database management systems that store sales transactions or other critical data, need full transactional service. Single-user operation is faster than multi-user operation, since no overhead is incurred by locking. Running with the recovery system disabled is faster than running with it enabled, since log records need not be written when changes are made to the database.

In addition, some core subsystems, including the locking system and the logging facility, can be used outside the context of the access methods as well. Although few users have chosen to do so, it is possible to use only the lock manager in Berkeley DB to control concurrency in an application, without using any of the standard database services. Alternatively, the caller can integrate locking of non-database resources with Berkeley DB's transactional two-phase locking system, to impose transaction semantics on objects outside the database.

### 3.1. Programmatic interfaces

Berkeley DB defines a simple API for database management. The package does not include industry-standard programmatic interfaces such as Open Database Connectivity (ODBC), Object Linking and Embedding for Databases (OleDB), or Structured Query Language (SQL). These interfaces, while useful, were designed to promote interoperability of database systems, and not simplicity or performance.

In response to customer demand, Berkeley DB 2.5 introduced support for the XA standard [Open94]. XA permits Berkeley DB to participate in distributed transactions under a transaction processing monitor like Tuxedo from BEA Systems. Like XA, other standard interfaces can be built on top of the core system. The standards do not belong inside Berkeley DB, since not all applications need them.

### 3.2. Working with records

A database user may need to search for particular keys in a database, or may simply want to browse available records. Berkeley DB supports both keyed access, to find one or more records with a given key, or sequential access, to retrieve all the records in the database one at

a time. The order of the records returned during sequential scans depends on the access method. B+tree and Recno databases return records in sort order, and Hash databases return them in apparently random order. Similarly, Berkeley DB defines simple interfaces for inserting, updating, and deleting records in a database.

### 3.3. Long keys and values

Berkeley DB manages keys and values as large as  $2^{32}$  bytes. Since the time required to copy a record is proportional to its size, Berkeley DB includes interfaces that operate on partial records. If an application requires only part of a large record, it requests partial record retrieval, and receives just the bytes that it needs. The smaller copy saves both time and memory.

Berkeley DB allows the programmer to define the data types of keys and values. Developers use any type expressible in the programming language.

### 3.4. Large databases

A single database managed by Berkeley DB can be up to  $2^{48}$  bytes, or 256 petabytes, in size. Berkeley DB uses the host filesystem as the backing store for the database, so large databases require big file support from the operating system. Sleepycat Software has customers using Berkeley DB to manage single databases in excess of 100 gigabytes.

### 3.5. Main memory databases

Applications that do not require persistent storage can create databases that exist only in main memory. These databases bypass the overhead imposed by the I/O system altogether.

Some applications do need to use disk as a backing store, but run on machines with very large memory. Berkeley DB is able to manage very large shared memory regions for cached data pages, log records, and lock management. For example, the cache region used for data pages may be gigabytes in size, reducing the likelihood that any read operation will need to visit the disk in the steady state. The programmer declares the size of the cache region at startup.

Finally, many operating systems provide memory-mapped file services that are much faster than their general-purpose file system interfaces. Berkeley DB can memory-map its database files for read-only database use. The application operates on records stored directly on the pages, with no cache management overhead. Because the application gets pointers

directly into the Berkeley DB pages, writes cannot be permitted. Otherwise, changes could bypass the locking and logging systems, and software errors could corrupt the database. Read-only applications can use Berkeley DB's memory-mapped file service to improve performance on most architectures.

### 3.6. Configurable page size

Programmers declare the size of the pages used by their access methods when they create a database. Although Berkeley DB provides reasonable defaults, developers may override them to control system performance. Small pages reduce the number of records that fit on a single page. Fewer records on a page means that fewer records are locked when the page is locked, improving concurrency. The per-page overhead is proportionally higher with smaller pages, of course, but developers can trade off space for time as an application requires.

### 3.7. Small footprint

Berkeley DB is a compact system. The full package, including all access methods, recoverability, and transaction support is roughly 175K of text space on common architectures.

### 3.8. Cursors

In database terminology, a cursor is a pointer into an access method that can be called iteratively to return records in sequence. Berkeley DB includes cursor interfaces for all access methods. This permits, for example, users to traverse a B+tree and view records in order. Pointers to records in cursors are persistent, so that once fetched, a record may be updated in place. Finally, cursors support access to chains of duplicate data items in the various access methods.

### 3.9. Joins

In database terminology, a join is an operation that spans multiple separate tables (or in the case of Berkeley DB, multiple separate DB files). For example, a company may store information about its customers in one table and information about sales in another. An application will likely want to look up sales information by customer name; this requires matching records in the two tables that share a common customer ID field. This combining of records from multiple tables is called a join.

Berkeley DB includes interfaces for joining two or more tables.

### 3.10. Transactions

Transactions have four properties [Gray93]:

- They are atomic. That is, all of the changes made in a single transaction must be applied at the same instant or not at all. This permits, for example, the transfer of money between two accounts to be accomplished, by making the reduction of the balance in one account and the increase in the other into a single, atomic action.
- They must be consistent. That is, changes to the database by any transaction cannot leave the database in an illegal or corrupt state.
- They must be isolatable. Regardless of the number of users working in the database at the same time, every user must have the illusion that no other activity is going on.
- They must be durable. Even if the disk that stores the database is lost, it must be possible to recover the database to its last transaction-consistent state.

This combination of properties — atomicity, consistency, isolation, and durability — is referred to as ACIDity in the literature. Berkeley DB, like most database systems, provides ACIDity using a collection of core services.

Programmers can choose to use Berkeley DB's transaction services for applications that need them.

#### 3.10.1. Write-ahead logging

Programmers can enable the logging system when they start up Berkeley DB. During a transaction, the application makes a series of changes to the database. Each change is captured in a log entry, which holds the state of the database record both before and after the change. The log record is guaranteed to be flushed to stable storage before any of the changed data pages are written. This behavior — writing the log before the data pages — is called *write-ahead logging*.

At any time during the transaction, the application can *commit*, making the changes permanent, or *roll back*, cancelling all changes and restoring the database to its pre-transaction state. If the application rolls back the transaction, then the log holds the state of all changed pages prior to the transaction, and Berkeley DB simply restores that state. If the application commits the transaction, Berkeley DB writes the log records to disk. In-memory copies of the data pages already reflect the changes, and will be flushed as necessary during normal processing. Since log writes are sequential, but data page writes are random, this improves

performance.

#### 3.10.2. Crashes and recovery

Berkeley DB's write-ahead log is used by the transaction system to commit or roll back transactions. It also gives the recovery system the information that it needs to protect against data loss or corruption from crashes. Berkeley DB is able to survive application crashes, system crashes, and even catastrophic failures like the loss of a hard disk, without losing any data.

Surviving crashes requires data stored in several different places. During normal processing, Berkeley DB has copies of active log records and recently-used data pages in memory. Log records are flushed to the log disk when transactions commit. Data pages trickle out to the data disk as pages move through the buffer cache. Periodically, the system administrator backs up the data disk, creating a safe copy of the database at a particular instant. When the database is backed up, the log can be truncated. For maximum robustness, the log disk and data disk should be separate devices.

Different system failures can destroy memory, the log disk, or the data disk. Berkeley DB is able to survive the loss of any one of these repositories without losing any committed transactions.

If the computer's memory is lost, through an application or operating system crash, then the log holds all committed transactions. On restart, the recovery system rolls the log forward against the database, reapplying any changes to on-disk pages that were in memory at the time of the crash. Since the log contains pre- and post-change state for transactions, the recovery system also uses the log to restore any pages to their original state if they were modified by transactions that never committed.

If the data disk is lost, the system administrator can restore the most recent copy from backup. The recovery system will roll the entire log forward against the original database, reapplying all committed changes. When it finishes, the database will contain every change made by every transaction that ever committed.

If the log disk is lost, then the recovery system can use the in-memory copies of log entries to roll back any uncommitted transactions, flush all in-memory database pages to the data disk, and shut down gracefully. At that point, the system administrator can back up the database disk, install a new log disk, and restart the system.

### 3.10.3. Checkpoints

Berkeley DB includes a checkpointing service that interacts with the recovery system. During normal processing, both the log and the database are changing continually. At any given instant, the on-disk versions of the two are not guaranteed to be consistent. The log probably contains changes that are not yet in the database.

When an application makes a *checkpoint*, all committed changes in the log up to that point are guaranteed to be present on the data disk, too. Checkpointing is moderately expensive during normal processing, but limits the time spent recovering from crashes.

After an application or operating system crash, the recovery system only needs to go back two checkpoints<sup>1</sup> to start rolling the log forward. Without checkpoints, there is no way to be sure how long restarting after a crash will take. With checkpoints, the restart interval can be fixed by the programmer. Recovery processing can be guaranteed to complete in a second or two.

Software crashes are much more common than disk failures. Many developers want to guarantee that software bugs do not destroy data, but are willing to restore from tape, and to tolerate a day or two of lost work, in the unlikely event of a disk crash. With Berkeley DB, programmers may truncate the log at checkpoints. As long as the two most recent checkpoints are present, the recovery system can guarantee that no committed transactions are lost after a software crash. In this case, the recovery system does not require that the log and the data be on separate devices, although separating them can still improve performance by spreading out writes.

### 3.10.4. Two-phase locking

Berkeley DB provides a service known as two-phase locking. In order to reduce the likelihood of deadlocks and to guarantee ACID properties, database systems manage locks in two phases. First, during the operation of a transaction, they acquire locks, but never release them. Second, at the end of the transaction, they release locks, but never acquire them. In practice, most database systems, including Berkeley DB, acquire locks on demand over the course of the transaction, then flush the log, then release all locks.

<sup>1</sup> One checkpoint is not far enough. The recovery system cannot be sure that the most recent checkpoint completed — it may have been interrupted by the crash that forced the recovery system to run in the first place.

Berkeley DB can lock entire database files, which correspond to tables, or individual pages in them. It does no record-level locking. By shrinking the page size, however, developers can guarantee that every page holds only a small number of records. This reduces contention.

If locking is enabled, then read and write operations on a database acquire two-phase locks, which are held until the transaction completes. Which objects are locked and the order of lock acquisition depend on the workload for each transaction. It is possible for two or more transactions to deadlock, so that each is waiting for a lock that is held by another.

Berkeley DB detects deadlocks and automatically rolls back one of the transactions. This releases the locks that it held and allows the other transactions to continue. The caller is notified that its transaction did not complete, and may restart it. Developers can specify the deadlock detection interval and the policy to use in choosing a transaction to roll back.

The two-phase locking interfaces are separately callable by applications that link Berkeley DB, though few users have needed to use that facility directly. Using these interfaces, Berkeley DB provides a fast, platform-portable locking system for general-purpose use. It also lets users include non-database objects in a database transaction, by controlling access to them exactly as if they were inside the database.

The Berkeley DB two-phase locking facility is built on the fastest correct locking primitives that are supported by the underlying architecture. In the current implementation, this means that the locking system is different on the various UNIX platforms, and is still more different on Windows NT. In our experience, the most difficult aspect of performance tuning is finding the fastest locking primitives that work correctly on a particular architecture and then integrating the new interface with the several that we already support.

The world would be a better place if the operating systems community would uniformly implement POSIX locking primitives and would guarantee that acquiring an uncontested lock was a fast operation. Locks must work both among threads in a single process and among processes.

## 3.11. Concurrency

Good performance under concurrent operation is a critical design point for Berkeley DB. Although Berkeley DB is itself not multi-threaded, it is thread-safe, and runs well in threaded applications. Philosophically, we view the use of threads and the choice of a threads

package as a policy decision, and prefer to offer mechanism (the ability to run threaded or not), allowing applications to choose their own policies.

The locking, logging, and buffer pool subsystems all use shared memory or other OS-specific sharing facilities to communicate. Locks, buffer pool fetches, and log writes behave in the same way across threads in a single process as they do across different processes on a single machine.

As a result, concurrent database applications may start up a new process for every single user, may create a single server which spawns a new thread for every client request, or may choose any policy in between.

Berkeley DB has been carefully designed to minimize contention and maximize concurrency. The cache manager allows all threads or processes to benefit from I/O done by one. Shared resources must sometimes be locked for exclusive access by one thread of control. We have kept critical sections small, and are careful not to hold critical resource locks across system calls that could deschedule the locking thread or process. Sleepycat Software has customers with hundreds of concurrent users working on a single database in production.

## 4. Engineering Philosophy

Fundamentally, Berkeley DB is a collection of access methods with important facilities, like logging, locking, and transactional access underlying them. In both the research and the commercial world, the techniques for building systems like Berkeley DB have been well-known for a long time.

The key advantage of Berkeley DB is the careful attention that has been paid to engineering details throughout its life. We have carefully designed the system so that the core facilities, like locking and I/O, surface the right interfaces and are otherwise opaque to the caller. As programmers, we understand the value of simplicity and have worked hard to simplify the interfaces we surface to users of the database system.

Berkeley DB avoids limits in the code. It places no practical limit on the size of keys, values, or databases; they may grow to occupy the available storage space.

The locking and logging subsystems have been carefully crafted to reduce contention and improve throughput by shrinking or eliminating critical sections, and reducing the sizes of locked regions and log entries.

There is nothing in the design or implementation of Berkeley DB that pushes the state of the art in database systems. Rather, we have been very careful to get the engineering right. The result is a system that is

superior, as an embedded database system, to any other solution available.

Most database systems trade off simplicity for correctness. Either the system is easy to use, or it supports concurrent use and survives system failures. Berkeley DB, because of its careful design and implementation, offers both simplicity and correctness.

The system has a small footprint, makes simple operations simple to carry out (inserting a new record takes just a few lines of code), and behaves correctly in the face of heavy concurrent use, system crashes, and even catastrophic failures like loss of a hard disk.

## 5. The Berkeley DB 2.x Distribution

Berkeley DB is distributed in source code form from [www.sleepycat.com](http://www.sleepycat.com). Users are free to download and build the software, and to use it in their applications.

### 5.1. What is in the distribution

The distribution is a compressed archive file. It includes the source code for the Berkeley DB library, as well as documentation, test suites, and supporting utilities.

The source code includes build support for all supported platforms. On UNIX systems Berkeley DB uses the GNU autoconfiguration tool, `autoconf`, to identify the system and to build the library and supporting utilities. Berkeley DB includes specific build environments for other platforms, such as VMS and Windows.

#### 5.1.1. Documentation

The distributed system includes documentation in HTML format. The documentation is in two parts: a UNIX-style reference manual for use by programmers, and a reference guide which is tutorial in nature.

#### 5.1.2. Test suite

The software also includes a complete test suite, written in Tcl. We believe that the test suite is a key advantage of Berkeley DB over comparable systems.

First, the test suite allows users who download and build the software to be sure that it is operating correctly.

Second, the test suite allows us, like other commercial developers of database software, to exercise the system thoroughly at every release. When we learn of new bugs, we add them to the test suite. We run the test suite continually during development cycles, and

always prior to release. The result is a much more reliable system by the time it reaches beta release.

## 5.2. Binary distribution

Sleepycat makes compiled libraries and general binary distributions available to customers for a fee.

## 5.3. Supported platforms

Berkeley DB runs on any operating system with a POSIX 1003.1 interface [IEEE96], which includes virtually every UNIX system. In addition, the software runs on VMS, Windows/95, Windows/98, and Windows/NT. Sleepycat Software no longer supports deployment on sixteen-bit Windows systems.

## 6. Berkeley DB 2.x Licensing

Berkeley DB 2.x is distributed as an Open Source product. The software is freely available from us at our Web site, and in other media. Users are free to download the software and build applications with it.

The 1.x versions of Berkeley DB were covered by the UC Berkeley copyright that covers software freely redistributable in source form. When Sleepycat Software was formed, we needed to draft a license consistent with the copyright governing the existing, older software. Because of important differences between the UC Berkeley copyright and the GPL, it was impossible for us to use the GPL. A second copyright, with terms contradictory to the first, simply would not have worked.

Sleepycat wanted to continue Open Source development of Berkeley DB for several reasons. We agree with Raymond [Raym98] and others that Open Source software is typically of higher quality than proprietary, binary-only products. Our customers benefit from a community of developers who know and use Berkeley DB, and can help with application design, debugging, and performance tuning. Widespread distribution and use of the source code tends to isolate bugs early, and to get fixes back into the distributed system quickly. As a result, Berkeley DB is more reliable. Just as importantly, individual users are able to contribute new features and performance enhancements, to the benefit of everyone who uses Berkeley DB. From a business perspective, Open Source and free distribution of the software creates share for us, and gives us a market into which we can sell products and services. Finally, making the source code freely available reduces our support load, since customers can find and fix bugs without recourse to us, in many cases.

To preserve the Open Source heritage of the older Berkeley DB code, we drafted a new license governing the distribution of Berkeley DB 2.x. We adopted terms from the GPL that make it impossible to turn our Open Source code into proprietary code owned by someone else.

Briefly, the terms governing the use and distribution of Berkeley DB are:

- your application must be internal to your site, or
- your application must be freely redistributable in source form, or
- you must get a license from us.

For customers who prefer not to distribute Open Source products, we sell licenses to use and extend Berkeley DB at a reasonable cost.

We work hard to accommodate the needs of the Open Source community. For example, we have crafted special licensing arrangements with Gnome to encourage its use and distribution of Berkeley DB.

Berkeley DB conforms to the Open Source definition [Open99]. The license has been carefully crafted to keep the product available as an Open Source offering, while providing enough of a return on our investment to fund continued development and support of the product. The current license has created a business capable of funding three years of development on the software that simply would not have happened otherwise.

## 7. Summary

Berkeley DB offers a unique collection of features, targeted squarely at software developers who need simple, reliable database management services in their applications. Good design and implementation and careful engineering throughout make the software better than many other systems.

Berkeley DB is an Open Source product, available at [www.sleepycat.com](http://www.sleepycat.com) for download. The distributed system includes everything needed to build and deploy the software or to port it to new systems.

Sleepycat Software distributes Berkeley DB under a license agreement that draws on both the UC Berkeley copyright and the GPL. The license guarantees that Berkeley DB will remain an Open Source product and provides Sleepycat with opportunities to make money to fund continued development on the software.

## 8. References

[Come79]

Comer, D., "The Ubiquitous B-tree," *ACM Computing Surveys* Volume 11, number 2, June 1979.

[Gray93]

Gray, J., and Reuter, A., *Transaction Processing: Concepts and Techniques*, Morgan-Kaufman Publishers, 1993.

[IEEE96]

Institute for Electrical and Electronics Engineers, *IEEE/ANSI Std 1003.1*, 1996 Edition.

[Litw80]

Litwin, W., "Linear Hashing: A New Tool for File and Table Addressing," *Proceedings of the 6th International Conference on Very Large Databases (VLDB)*, Montreal, Quebec, Canada, October 1980.

[Open94]

The Open Group, *Distributed TP: The XA+ Specification, Version 2*, The Open Group, 1994.

[Open99]

Opensource.org, "Open Source Definition," [www.opensource.org/osd.html](http://www.opensource.org/osd.html), version 1.4, 1999.

[Raym98]

Raymond, E.S., "The Cathedral and the Bazaar," [www.tuxedo.org/~esr/writings/cathedral-bazaar/cathedral-bazaar.html](http://www.tuxedo.org/~esr/writings/cathedral-bazaar/cathedral-bazaar.html), January 1998.

[Selt91]

Seltzer, M., and Yigit, O., "A New Hashing Package for UNIX," *Proceedings 1991 Winter USENIX Conference*, Dallas, TX, January 1991.

[Selt92]

Seltzer, M., and Olson, M., "LIBTP: Portable Modular Transactions for UNIX," *Proceedings 1992 Winter Usenix Conference*, San Francisco, CA, January 1992.

[Ston82]

Stonebraker, M., Stettner, H., Kalash, J., Guttman, A., and Lynn, N., "Document Processing in a Relational Database System," Memorandum No. UCB/ERL M82/32, University of California at Berkeley, Berkeley, CA, May 1982.



# The FreeBSD Ports Collection

Satoshi Asami, The FreeBSD Project

asami@freebsd.org

<http://www.freebsd.org/ports/>

## Overview

FreeBSD is an open source operating system based on 4.4BSD-Lite2, a version of UNIX from the University of California at Berkeley. It is maintained by a group of volunteers from around the world. In addition to providing a complete operating system, the FreeBSD project supports an extensive collection of sanctioned third-party software called *the Ports Collection*, many of which were contributed by the users. In addition to the source form, most of the ports are provided as binary packages too.

## Port Skeletons and Distfiles

The Ports Collection consists of a single make macro file, `bsd.port.mk`, and some skeleton files for each port that describe how to compile and install the software. If there were changes made to the original software to compile it on FreeBSD, patches to reproduce those changes are included too.

One of the items specified in the port's `Makefile` is the name and URL where the source files ("*distfiles*") of the original software are located. When the user attempts to compile a port, they are fetched over the Internet if they do not exist on the system. The distfiles are checksum-verified to ensure consistency, as well as guarding against possible Trojan horse attacks.

Since the distfiles can be fetched on demand, this allows the Ports Collection itself to stay small. For instance, with over 2,200 ports as of April 1999, all the skeleton files total only about 70MB. In contrast, the entire set of distfiles, most of which are compressed archive files, are over 1.4GB. It takes over 8GB to extract and compile them all at once.

## Packages

In addition to providing an easy way to compile programs, `bsd.port.mk` provides a set of commands to create binary *packages* of installed ports. These packages, which are compressed archive files with some additional information, can be installed on a FreeBSD system using the `pkg_add` command. They contain a listing of the entire set of installed files, so they can also be deleted completely, using the `pkg_delete` command. Each release

of FreeBSD ships with a complete set of packages. Currently, there are about 1.2GB of packages.

The Ports Collection framework always supported a simple top-down build of packages. In other words, when the user types "make package" at the root directory of the ports hierarchy, `bsd.port.mk` will arrange for the build process to go into every single subdirectory and build packages for each of them, one by one. This method, akin to the way many large software trees are built, has exhibited many problems as the Ports Collection grew.

## Dependencies

Many software depend on others to build or run. These are called *dependencies*. If port A requires port B, then port A is called the *dependent* port and port B is the *dependency*. The Ports Collection has a mechanism of handling dependencies automatically. When a dependent port is built, the environment is checked to see if the dependency is already installed, and if not, the dependency is built and installed first. The dependency check is recursive, so if a dependency requires another port, it will also be checked and built. This chain can continue to an arbitrary depth.

The dependency information is also recorded in packages. When a package is installed, `pkg_add` checks if all the dependencies are installed as well, and if not, they will be installed automatically. Again, package dependency checking is recursive, and installing one package could potentially pull in dozens of other packages.

One of the interesting applications of the dependency mechanism is to create *meta-packages*, i.e., empty packages that make it easy for users to install several packages at once.

## Problems

There were many issues that had to be addressed as the Ports Collection grew from less than 200 ports in January 1995 to over 2,200 ports in April 1999. In addition to providing a brief summary of the history of the Ports Collection, this talk addresses the problems, such as shared library conflicts, dependency detection, etc., and how we resolved them. I will also describe the process we use to build the 2,000 packages in a few hours before releases.



# Multilingual vi clones: past, now and the future

Jun-ichiro itojun Hagino/KAME Project  
`itojun@{iijlab,kame}.net`

Yoshitaka Tokugawa/WIDE Project

## Outline

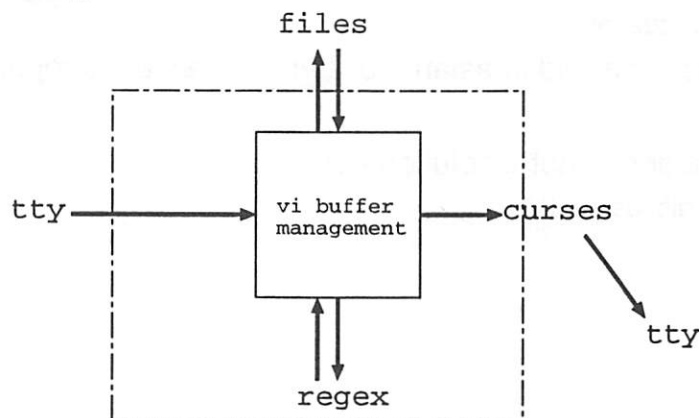
- Internal structures and issues in:
  - Japanized elvis
  - Multilingual nvi
- Experiences gained in asian multibyte characters support
- Note: Unicode is not a solution here
  - to be discussed later

## Assumptions in normal vi/vi clones

- ASCII (7bit) only, 8bit chars just go through
  - The terminal software defines interpretation
- One byte occupies 1 column on screen (except tabs)
- Assumes western languages - space between words

## Architecture of normal vi

- tty input, filesystem, tty output (curses), vi internal buffer use the same encoding



## Western character encodings

- Character encoding and the language => assumptions

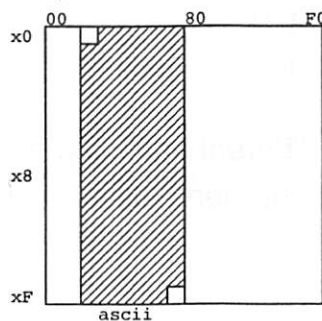
- Single byte encodings

- "ASCII" encoding

- ASCII character set: 94 characters

- Latin 1 encoding:

- ASCII character set
- iso-8859-1 character set, shifted 0x80



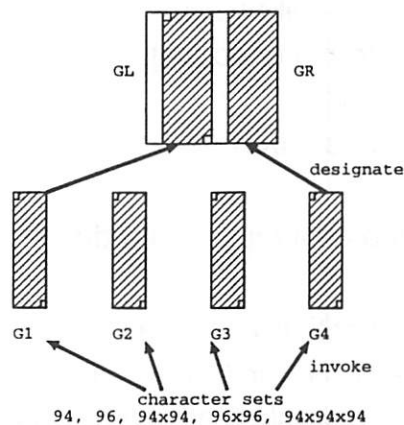
## ISO-2022 system

- Extensible character encoding system

- By switching multiple character sets by escape sequences
- Character set contains 94, 96, 94x94, 96x96, 94x94x94 chars

- ISO-2022 subset encodings are everywhere

- Latin 1: fixed mapping with ASCII and iso-8859-1
- X11 ctext



## Japanese encodings

A	B	C	漢	字	1
---	---	---	---	---	---

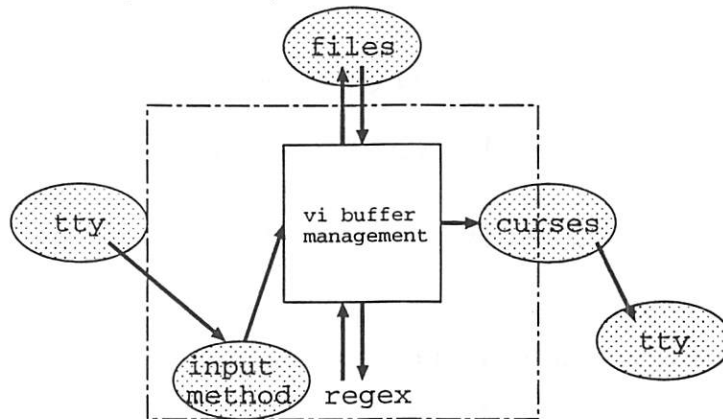
- ☐ JIS X0208 character set: 94x94 characters
- ☐ iso-2022-jp: Internet emails/netnews, UNIX
  - 41 42 43 1B 24 42 34 41 3B 7A 1B 28 42 31
- ☐ euc-jp: UNIX and other places
  - 41 42 43 B4 C1 BB FA 31
- ☐ sjis: MS-DOS and Macintosh community
  - 41 42 43 8A BF 8E 9A 31
  - Not an ISO-2022 variant
- ☐ Same character sets, different encoding method
- ☐ Single encoding is not sufficient - they all are used in various places!

## Asian people needs multibyte/multilingual support

- ☐ Multibyte character sets support
  - 2 or more byte/letter
- ☐ Byte width != character width on screen
- ☐ Input methods: ondemand conversion from ASCII to multibytes
  - Use third-party libraries, like Canna or Wnn
- ☐ Switching various external encoding methods
  - For file and terminal I/O
- ☐ **Seamless** multilingual support
- ☐ => Clarify/remove the assumptions made in vi implementation
- ☐ European people benefits from this as well
  - Handle iso-8859-x, koi8-r, and others in proper way
- ☐ Multilingual is more desirable than monolingual (Japanize)
  - Maintenance issues

## architecture of multilingual vi

- Can't assume single encoding
- Need input method (inside or outside vi)
- Must be able to switch encodings
  - tty input, input method, filesystem, tty output can use different encoding
- Internal encoding is the key issue



## Design goals: What is "seamless"?

- No "Chinese mode" nor "Japanese mode" in the editing session
- Any character set can be mixed in a text, without twist
  - Some of character encodings can accomodate, say Chinese, Korean and Japanese character sets at the same time
  - Mixed language texts - Chinese document annotated with Japanese
- Preserves information in the file
  - No implicit conversion/translation
  - Implicit conversion confuses user, and it does not match the vi design
  - If you need conversion, use : !
- Behaves just like normal vi, over multilingual characters
  - regex, cursor movement, whatever

## "jelvis" - Japanized elvis

- ☐ First generation of implementation
- ☐ Based on elvis by Steve Kirkendall
- ☐ Internal encoding: euc-jp
- ☐ External encoding: iso-2022-jp, euc-jp, sjis
- ☐ Internal encoding: 41 42 43 B4 C1 BB FA 31  

A	B	C	漢	字	1
---	---	---	---	---	---
- ☐ Internal encoding bytewidth == screen width
  - 2 bytes, 2 columns
- ☐ Maintenance/synchronization problem with kelvis/celvis
  - => Multilingual implementation is desirable

## "nvi-m17n" - multilingualized nvi

- ☐ Current generation of implementation
- ☐ Based on nvi by Keith Bostic
- ☐ Internal encoding: internal multibyte encoding
  - ASCII is 1 byte
  - 0x80-0xff are "multibyte tag" character
  - This is similar to Mule (multilingual emacs)
- ☐ External encoding: any of iso-2022 variants, and others
- ☐ Internal encoding: 41 42 43 88 34 41 88 3B 7A 31
  - "88" is the tag for JIS X0208 Kanji character set

A	B	C	漢	字	1
---	---	---	---	---	---
- ☐ Internal encoding bytewidth != screen width

## Additional features

- Switching I/O encoding:
  - `:set fileencoding=iso-2022-jp`
  - `:set inputencoding=big5`
  - `:set displayencoding=euc-tw`
- Input method support: "Canna" library from NEC
  - `:set cannaserver=server.itojun.org`
  - `:set cannakey=^O`

## Word boundary issues

- Asian words are not separated by spaces!
- Define word movement over Asian characters
  - The exact "word" movement requires syntactic analysis and dictionary lookup (very hard)
- Define character classes
  - Kanji letters, hiragana letters, western, symbols
- Define movement over word boundary
- Solves problem for most of the cases

萩野はFreenix会場にいます!!
- Need for explicit language information

## Regex library

- ☐ Some of regex library uses  $2^7$  as flag bit
  - Separate flag bit from the characters
- ☐ Character range ([a-z0-9]) as bitmap
  - Impossible for multibyte chars/multilingual internal code
  - Bitmap for ASCII, start-end for others
- ☐ Metacharacter (.) must match against single multibyte char

## Curses library

- ☐ Store character set information into screen buffer
  - ☐ Render accordingly on redraw
    - Character set
    - Character data (multibyte)
    - Offset from the beginning of the glyph
  - ☐ Multi-width characters support
    - Need to erase right half, when left half is overwritten
- |   |   |   |   |   |   |
|---|---|---|---|---|---|
| A | B | C | 漢 | 字 | 1 |
|---|---|---|---|---|---|
- ☐ Multibyte with `addch()` is cumbersome, use `addstr()`
    - Intermediate state is hard to manage

## Unicode as internal encoding?

- Unicode characteristics:
  - Well documented external multibyte encoding (UTF8/16)
  - 16 or 32bit fixed wide char for internal encoding (UCS2/4)
- Asian characters are "unified"
  - Some of Chinese/Korean/Japanese characters are mapped into single Unicode codepoint
  - As different characters are mapped into single codepoint, information will be lost (inverse conversion is impossible)
  - Language tagging -> "fixed-width wide char" is impossible
- Unicode is useful for "monolingual" asian processing
  - For example, ASCII + Chinese only
  - Or, modal support like "Chinese mode" or "Korean mode"
- Unicode is not useful for multilingual processing
- Additional Unicode support would be good
  - Unicode as a character set we support, not as the internal encoding

## nvi-m17n: next generation

- Use wide char (wchar\_t) for internal code
  - ISO/JIS standards suggest wide char
  - Memory is now cheap
- Can't really rely upon vendor's locale library
  - Too little support for stateful multibyte encodings
- Need massive modification to various places
  - Support for multiple encoding in locale library
  - Support for wide char in curses/regex/whatever
- Feedback modified locale library to the community
- Add Unicode support
  - Supply file converter as external tool

## Wide character library: status

- ☐ Wide char library is not really ready
  - curses, regex
  - Need support for column width query (for curses)
- ☐ Bugs in vendor-supplied locale library
  - Not heavily tested?
- ☐ Changing from char to wchar\_t is a big leap for the source code tree
- ☐ glibc
  - Assumes Unicode (no support for stateful encodings), single encoding in a program
- ☐ runelocale library
  - Encoding switchable by `$LANG`, no support for stateful encodings, single encoding in a program

## Observation

- ☐ Normal vi
  - 1byte/char
  - Single encoding (= ASCII)
- ☐ Japanized vi (jelvis)
  - Multibyte/char, bytewidth == width on screen
  - Multiple encoding in a program
- ☐ Multilingual vi (nvi-m17n)
  - Multibyte/char, bytewidth != width on screen
  - Multiple encoding in a program
- ☐ Next multilingual vi
  - Wide char, bytewidth != width on screen
  - multiple encoding in a program
- ☐ Multilingualization = less assumptions!

## Future work

- ☐ Provide modified runelocale library separately to \*BSD
- ☐ Right-to-left languages
- ☐ Support for other input method: cWnn (Chinese Wnn)

## References

- ☐ mailing list: `nvi-m17n@foretune.co.jp`
  - discussions are (at this moment) mainly in Japanese language, questions in English are welcome
- ☐ `ftp://ftp.foretune.co.jp/pub/tools/jelvis/`
- ☐ `ftp://ftp.foretune.co.jp/pub/tools/nvi-m17n/`
- ☐ Ken Lunde, "CJKV information processing", O'reilly



# Improving Application Performance through Swap Compression \*

R. Cervera   T. Cortes   Y. Becerra

*Departament d'Arquitectura de Computadors  
Universitat Politècnica de Catalunya - Barcelona*

<http://www.ac.upc.es/hpc>  
{rcervera,toni,yolandab}@ac.upc.es

## Abstract

The performance of large applications tends to be poor due to the high overhead added by the swapping mechanism. The same problem may be found in highly-loaded multi-programmed systems where many of the running applications have to use the swap space in order to be able to execute at the same time. Furthermore, those large applications might not be able to run on laptop or home computers as their resources are usually smaller than the ones found in an office system. In this paper, we present a solution to both problems that we have implemented in the Linux kernel. The idea consists of compressing the swapped pages and keeping them in a swap cache whenever possible. We have tested this new mechanism with a set of real applications obtaining a significant performance improvement.

## 1 Introduction

There are many applications that use large amounts of memory. These large applications take advantage of the swapping mechanism to run on the system as the available physical memory is not enough for them to run [12, 10]. The same problem appears when we try to run, on a laptop, the same applications we run on a desktop computer. These applications will rely on the swapping mechanism as laptop computers usually have less physical memory than desktop ones. Finally, multi-user environments tend to be very loaded and their applications have to swap out part of their memory so that all appli-

cations can run concurrently [16]. In all these cases, the performance of the applications is much lower than the one they would achieve if no swapping was needed. This happens because the swapping mechanism has to access the disk to keep the pages that do not fit in memory. It is clear that these applications, and the whole system, would benefit from a faster swapping system.

If we examine the same problem from a different point of view, we observe that increasing the number of pages that fit in the swap space without increasing the number of blocks in the swap partition would also be quite beneficial. We could run the same applications on a laptop than on a desktop system. Remember that laptops also have smaller disks if compared to desktop ones. This increase in swap space would also help multi-user systems to avoid getting out of memory. Finally, out-of-core applications could be programmed more easily as the global-memory restriction would not be so important.

Now a days it is quite normal to continue the office work at home. This usually means the use of large applications on a Linux box. These large applications fit well in the office machines but are too large to run efficiently on a smaller Linux box. In these cases, a fast swapping mechanism would be very beneficial as those applications would run faster and working at home would be less "painful". Furthermore, increasing the swap space at no cost would allow these kind of users to run applications that would normally not fit in their home machines.

These performance and space problems have motivated this work and its objectives. The first, and most important, objective is to speedup the swap mechanism. This will increase the performance of the applications that, for whatever reason, have to

---

\*This work has been supported by the Spanish Ministry of Education (CICYT) under the TIC-95-0429 and the TIC-94-0439 contracts.

keep part of their memory in the swap space. It is also an objective of this paper to increase the size of the memory offered to the applications without increasing the number of disk blocks in the swap partition. It is important to notice that should these two objectives be in conflict, we will favor performance over capacity. Finally, we want to achieve both improvements with the minimum number of changes in the original Linux kernel.

The main idea used to accomplish both objectives consists of compressing the pages that have to be swapped out. This will increase the number of pages that can be placed in the swap partition. Furthermore, it will also allow us to build a cache of compressed pages that will decrease the number of times the system has to access the swap device. It is important to notice that previous studies show that good compression ratios can be achieved when compressing memory pages [7]. The idea we present in this paper is similar, in essence, to the one proposed by Douglass [4], but some improvements and modifications have been done (see Section 5). We believe that now is a good time to reevaluate the results obtained in this previous work as the technology has improved significantly which means that compressing and decompressing pages can be done much more efficiently.

This paper is divided into 6 sections. In Section 2, we describe the concepts and ideas in which this work has been based. In this section, we also present some preliminary results that will lead the final design. Section 3 gives a detailed overview of the way the mechanism works. Section 4 presents the benchmarks used and the results obtained while running them on our system. In Section 5, we present the most significant work already done in the area. Finally, Section 6 presents the main conclusions that can be extracted from this paper.

## 2 General Ideas & First Results

### 2.1 Caching

It has already been proved that caching is a good way to increase the performance of disk operations [13]. In our scenario, a cache for swapped pages should also increase the swapping performance if a few problems can be solved. One such

cache would decrease the number of disk reads as some of the requested pages might be found in the cache. Swapping out pages could also take advantage of the cache as a swapped-out page might be freed before reaching the disk. Furthermore, if the pages have to go to the disk, the system could write many of these pages together in a single request. If we can write all of them sequentially in the disk, we will only have to pay the seek/search latency once per write instead of once per page.

Before we continue, it is a good time to go through some terminology that will be helpful throughout the rest of the paper.

**Page:** The virtual memory of applications is divided into portions of 4Kbytes. Each of these portions is known as a page.

**Buffer:** A buffer or cache buffer is a portion of 4Kbytes of memory where pages are stored before they are sent to the disk.

**Disk block:** This term refers to the disk portion where the information of a buffer is stored. This means that disk blocks will also be 4Kbytes in size. We should take in mind that this term does not refer to sectors nor file-system blocks.

### 2.2 Compressing Cached Pages

Adding a cache to the swapping mechanism means that some memory available for processes is now taken away for the cache. This means that the applications will have less memory to work with. If nothing else is done, we have only taken some fast memory from the applications to offer the same amount of memory but somewhat slower. This does not seem to be the solution to increase the performance of the applications. The ideal solution would be to take some fast memory from the users to offer them a somewhat slower but 2 or 3 times larger one. Of course, this new memory has to be faster than the disk. This would reduce the number of times the system has to access the disk for paging reasons. This can be achieved by compressing the swapped pages. In a compressed cache, the system can keep more pages than the ones taken from the applications.

Whenever a page is swapped out, the system compresses it before storing it in the cache. On the

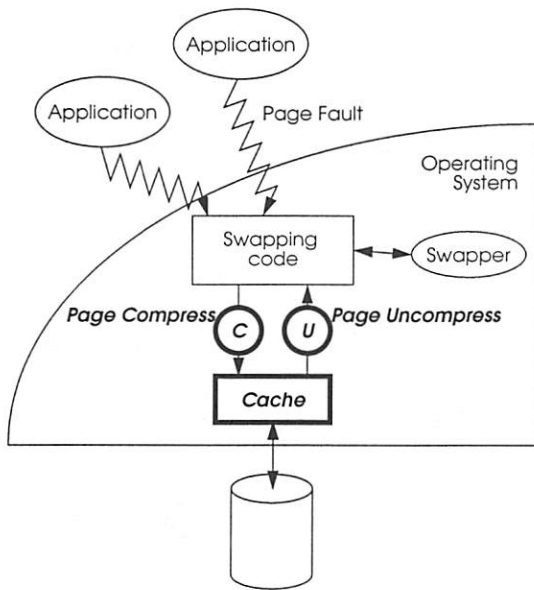


Figure 1: Conceptual vision of the compression and cache mechanism.

other hand, when the swap module requests a page, the system gets it either from the cache or the disk and decompresses it before handling it to the swap module. Figure 1 shows the first version of the path proposed for swapping in/out pages.

## 2.3 Batching Multiple Pages Together

A second advantage offered by the cache is the possibility of batching multiple pages together to write them contiguously to disk. This idea was first proposed in the VAX/VMS operating system [9]. This can be easily implemented as the system does not need to decide the physical location of a set of pages until they are really sent to the disk. Furthermore, as the pages are compressed, many more pages are written in a single disk write, thus decreasing the time spent on disk accesses.

## 2.4 Read/Write Path

Using all the proposed ideas, we built a preliminary prototype and we performed some measures and statistics. One of the most interesting results we obtained was the distribution of the two possible read-hit types: *read hit due write* and *read hits due write*.

**Read hit due write:** this kind of hits appear when the page is in the cache because it has recently been swapped out but has not yet been discarded. This means that the page is requested short after it was swapped out.

**Read hits due read:** this occurs when the page just requested is in a buffer that has recently been fetched from the disk. This means that another page, in the same disk block, has also been recently requested.

While examining both kind of hits, we detected that most of them were *hits due write*. This happens because the order in which pages are swapped out is not the same as the order in which they are swapped in. This led us to study the idea of not placing read buffers into the cache. This would allow recently written buffers to stay longer in the cache which might increase the hit ratio. Furthermore, this will also increase the write performance as less blocks will have to be sent to the disk.

In order to examine the effect of not placing read buffers in the cache, we implemented two versions of the preliminary prototype. A first one where the read buffers were placed in the cache and a second one where they were not. After running a set of benchmarks in both prototypes, we observed that the difference in the number of hits obtained by both systems was quite similar in most cases [3]. Furthermore, we also observed that the number of disk writes performed when reads do not interfere the cache is much lower than when reads are placed in the cache. This should increase the performance of the system as less writes are done and a similar number of reads are needed (similar read hit ratio).

Not placing read buffers in the cache has another interesting side effect. As reads do not need to make room in the cache, they will never have to perform a write operation to clean a dirty buffer. This will avoid many disk accesses while swapping in pages.

After this modification, the read disk blocks will not be placed into the cache. This does not mean that swapping-in operations will not take advantage of the cache. They will first try to find the page in the cache as it might have recently been written (*read hit due write*). If it is not in the cache, then the system will read the page, decompress it and forget about the rest of pages stored in the same disk block. Figure 2 shows the new path for swapping pages in and out.

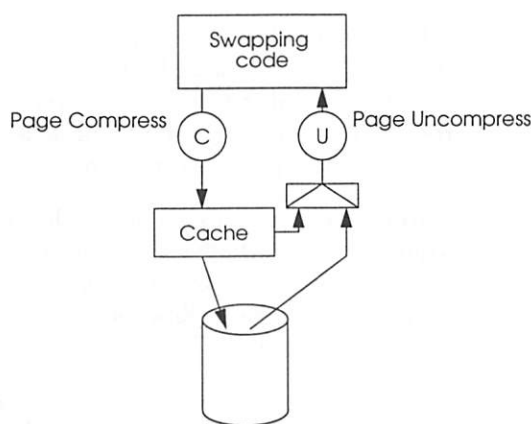


Figure 2: New swapping path where swapped-in pages are not kept in the cache.

Finally, another important side effect of not caching read requests is a simplification on the code. We will not get into many details now, but it is clear that a sapping-in operation will only have to search the page in the cache or to read it from the disk. It will not have to worry about cleaning buffers from the cache and it will also avoid most of the locking problems.

### 3 Prototype Description

In this section, we will describe the most important operations, policies and algorithm used in the final prototype. We will start describing the data structures used and then, the four main operations will be explained with some detail. We have to keep in mind that only the main ideas are described and that technical issues such as locking or very infrequent situations are not presented in the paper.

Regarding some implementation details, the prototype has been built in the Linux operating system (kernel version 2.0.34) [2]. The compression algorithm chosen has been *lzo* [11] which is based on the Ziv-Lempel data compressor [17]. This algorithm was chosen as it obtained a good ratio between speed and compression. On one hand, it achieved compression ratios better than 50% in most of the experiments (Table 1). On the other hand, the average time needed to compress a 4Kbyte page is about 300 microseconds while the one needed to decompress a buffer is only about 50 microseconds<sup>1</sup>.

<sup>1</sup>Measured on a Pentium II at 350MHz

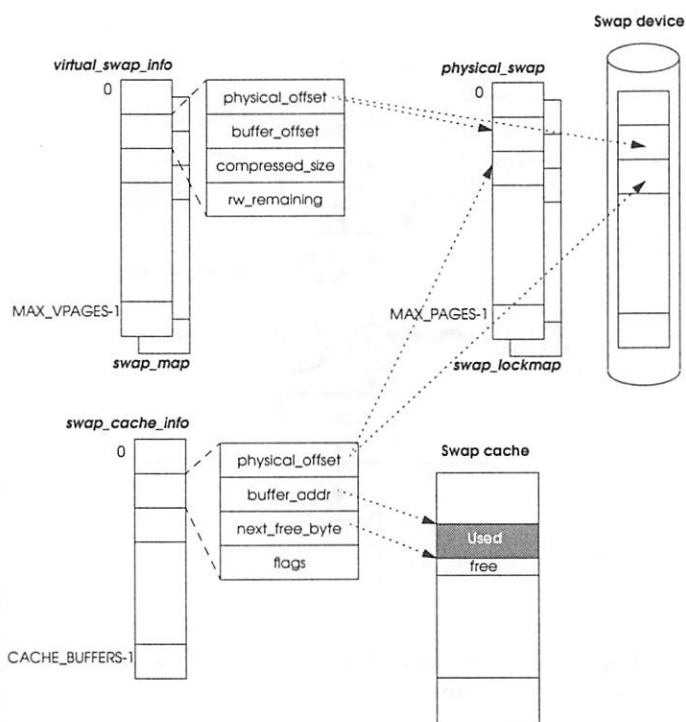


Figure 3: Data structures needed for the compressed swap.

#### 3.1 Data Structures

In order to implement this mechanism we have added some data structures to the original Linux kernel. In this section, we will describe each of these structures in some detail. A general picture with all the structures and most of the fields is presented in Figure 3.

- **virtual\_swap\_info**: this structure keeps the information of all the compressed pages. The size of this array is MAX\_VPAGES, which is the maximum number of compressed pages that our system will be able to handle. The size of this array can be modified to suit the needs of each system as explained in Section 3.6. In each entry of this array we have the following information:

**physical\_offset**: it indicates the disk block where the compressed page is stored.

**buffer\_offset**: it is the field used to mark the position where this compressed page is kept. As we store more than one compressed pages per disk block, we need to know at which byte does the page start.

**compressed\_size:** as can be guessed from its name, this field is used to keep the size of the swapped page once it has been compressed.

**rw\_remaining:** it is a counter of the number of pending read or write operations for this page. We need this information not to free a page while still being used.

- **swap\_map:** this structure was already used in the original Linux kernel. We have only modified its size as we need an entry for each one in the **virtual\_swap\_info** array. Its function is to keep the number of processes that have this page mapped in their address space.
- **physical\_swap:** for each disk block, we need to know the number of compressed pages kept in it. This table is responsible for maintaining this information.
- **swap\_lockmap:** there are situation where we need to perform atomic operations on the disk block. To ensure that no other process will work with a given block, we use this data structure already implemented in the original Linux kernel. It is a bitmap where each bit tells whether the given disk block is being used in exclusive mode or not.
- **swap\_cache\_info:** this structure is used to keep all the information needed to maintain the compressed cache. It has as many entries as buffers in the cache (**CACHE\_BUFFERS**).

**physical\_offset:** it indicates the disk block assigned to this buffer.

**buffer\_addr:** it points to the cache buffer where the compressed pages are really stored. We need this pointer as all buffers are not necessarily contiguous. This is because we cannot allocate as many buffers as needed in a single call (Linux implementation issues).

**next\_free\_byte:** it keeps the first free byte. This is the position where the next compressed page inserted in this buffer will be placed.

**flags:** cache buffers need some flags such as the dirty bit.

- **swap\_cache:** this structure is just a set of buffers (not necessarily contiguous) that are used to keep the compressed pages before they are sent to the disk.

- **Swap device:** finally, this is the disk partition where the compressed pages are finally stored.

### 3.2 Getting Free Space (**get\_swap\_page**)

In the original kernel, this function returned the disk offset where the page would be stored but this offset was used as an identifier for the swapped page. Only the swap code used it to access the disk. As we need to know the size of the compressed page before assigning it to a disk block (it has to fit in it), we will return an index to the **virtual\_swap\_info** table. To the rest of the kernel, this function behaves as always and the system believes that the swap partition is a larger one.

To return this index, the system searches for a free entry in the **virtual\_swap\_info** array. A given entry is free when no process is using it (**swap\_map[i] == 0**) and when there are no operations remaining to be done on this page (**virtual\_swap\_info[i].rw\_remaining == 0**).

Another important issues is that the system cannot return an index unless it is sure that there will be enough disk space to keep the page. For this reason, the system will always assume the worst case. Until it knows the real compressed size, the system will assume that the page needs a full disk block to be stored. As soon as the system knows its real size, it will update this information.

### 3.3 Freeing a Page (**swap\_free**)

Freeing a page consists of decrementing the number of processes that are using the page (**swap\_map[i]--**). Whenever this becomes zero, the system has to free this space from the cache buffer or the disk block. This is done by decrementing the number of compressed pages in its disk block or cache buffer (**physical\_swap[i]--**). Should this page be the last one in the buffer, the whole buffer should also be freed to be used by other pages in the future.

The above scenario is the best possible case. For mutual exclusion reasons, the order in which the operations are done can be broken and a page might be freed while a write operation is still pending. In this case, the page is marked to be freed as soon as

the pending operations are finished. We will see this while describing the swapping in and out operations.

Another important issue related to freeing a page is the recompectation of blocks and buffers. If the page that is being freed was stored in the middle of a block or buffer, we could think of reallocating all pages together to merge this new space with the free space already remaining in the block or buffer. Doing this on disk blocks is completely out of the question as it would mean reading the disk (too much overhead). Recompecting cache buffers is a feasible task but we have seen that it does not increase the performance of the system and makes the code more complex [3]. For these reasons, we will never reuse the space of a freed page until all the compressed pages in a block or buffer have been freed.

### 3.4 Swapping Out (rw\_swap\_page)

Once the system wants to swap out a page, it compresses the page and tries to write it into the cache. Performing this operation, we might find that there is not enough free space to cache this new page. This means that all buffers are dirty (and have not been sent to the disk) and that all of them have less free space than the size of the compressed page. It is important to notice that the system will never split a compressed page among several buffers. When the system runs out of free space in the compressed cache, it performs a cleaning operation. Once it is done, at least one buffer will not be dirty and the system will be able to use it to put the new compressed page.

As we mentioned when describing the free operation, there are cases where a page could not be freed because there was a write operation still pending. If this is the last pending write operation and the page is marked as to be freed, the system will free the page after the write has been finished. The same steps as in the original free operation are taken.

#### Cleaning Mechanism

To clean the cache we need to send one or more buffers to the disk. This will allow the system to reuse them as an up-to-date copy of the data will be kept in the swap device. The intuitive idea of cleaning the cache consists of sending to the disk all buffers when no free space is left. As buffers do

not usually get completely filled with compressed pages, we have modified the concept of full buffer as follows:

**Full buffers** are those ones that have less free space than the average size of the last 100 compressed pages.

Using this new concept, whenever a page does not fit in the cache, all *full buffers* will be sent to the disk in a single operation where all of them are written contiguously on the disk. This will increase the performance of the write operations significantly.

Should the system need to perform a clean operation when there are no *full buffers*, the buffer with more data will be sent to disk.

As we do not want to wait until no free space is left on the cache to clean it, whenever it has a given percentage of its buffers *full*, the system writes them to the disk. This percentage can be adjusted to the needs of the system as will be seen in a later section. This operation is currently done in a synchronous way but we are working to make it asynchronous.

An important detail is the addition of a flag that tells whether there is a cleaning operation already running. If this flag is on, a second concurrent cleaning will not be done as only one is really needed.

### 3.5 Swapping In (rw\_swap\_page)

This is the simplest operation. Whenever a page is requested, the system searches for it in the cache. If the page is found in the cache, the system decompresses it and places the result on the user address space. Otherwise, if the page is in the disk, the disk block is read and the page is decompressed as in the previous case. As read disk blocks are not placed in the cache, if another page from the same buffer is requested, a new disk read will be needed (remember that *hits\_do\_read* are not very frequent).

### 3.6 Driver to Modify the Parameters

To simplify the task of setting the parameters for the compressed swap we have also implemented a driver that allows the superuser to modify the following parameters when the swap is off.

**Virtual space size:** maximum number of compressed pages that the system will be able to handle (MAX\_VPAGES).

**Cache size:** the number of Kbytes used for caching.

**Cleaning threshold:** the percentage of full buffers the system needs to find to perform a cleaning operation.

## 4 Experimental Evaluation

### 4.1 Methodology

All the results presented in this paper have been measured on a Pentium II running at 350MHz. The amount of physical memory was 64 Mbytes, and the size of the swap partition was 128Mbytes. This partition was located on a Ultra-SCSI hard disk.

All the measures presented are the average of, at least, 10 executions, in single-user mode, where the best and worst ones have been discarded.

### 4.2 Benchmark Description

To measure the performance of this proposal, we need to see the effect it has on a set of benchmarks. Before getting into a detailed description of the benchmarks, we would like to describe the three characteristics a benchmark can have that may have a higher effect on the behavior of the proposed system.

**Concurrency.** It is important to see that the number of processes in the benchmark will affect the behavior of the system. If only one process is running in the system, the application that is swapping out pages will not have to wait for another application that may have locked some of the resources it needs. No other application will try to swap in/out pages.

**I/O.** Another benchmark parameter that will affect the system is the amount of file-system I/O performed by the benchmark. This I/O may conflict with the one performed by the paging

system because both are done in the same disk (although in different partitions).

**Compression ratio.**<sup>2</sup> Finally, the compression ratio may affect the system in two ways. First, the better pages compress, the larger the final size of the swap area will be. Second, if pages have a good compression ratio, the number of pages that can be kept in the cache will be higher. Thus the number of disk accesses should be lower than with bad compression ratios.

Once described the most important characteristics, we will describe the benchmarks used.

- **fft.** It executes a fast Fourier transformation with a 2048x2048 matrix. The values of the elements in the matrix are set randomly.
- **fft x10.** This benchmark is very similar to the previous one but 10 ffts are executed concurrently and the size of the matrixes is decreased to 512x512 elements.
- **sort.** In this benchmark, we perform an in-memory sort of a text file. The input file is build by appending the /usr/dict/word file many times and then unsorting it as much as possible.
- **sort x6.** In this benchmark 6 sorts are executed concurrently. The file to be sorted is built as the previous one but 5 times smaller to limit the execution time of the benchmark.
- **simulator.** A simulator of a network of disks currently being used in our research group. It is an event-based simulator that uses large amounts of memory. This memory compresses very well as many of the events in the queues have similar information. Furthermore, the memory library used, does not free the allocated memory after a `free`, the library keeps the memory block in a hash queue for further use. This "freed" memory is also easy to compress. Although the compression ratio of this application seems to be unrealistic, there are other applications in a typical Unix system that achieve compression ratios better than 10% such as `awk` [7].

<sup>2</sup> $compression\_ratio = compressed\_size/page\_size$ . This means that high percentages denote bad compression

	Concurrent processes	File I/O	Compress ratio
fft	1	none	64.9%
fft x10	10	none	61.2%
sort	1	start/end	46.5%
sort x6	6	start/end	51.3%
simulator	1	start/end	6.7%
simulator x5	5	start/end	1.1%
xanim	1	first part	27.8%
xanim x4	4	first part	37.9%

Table 1: Benchmark characteristics.

- **simulator x5.** Five concurrent executions of the simulator but with a smaller input.
- **xanim.** A visualization of a video file in avi format. This video is dithered using the Floyd-Steinberg algorithm. This means that the file has to be decompressed in memory to do the dithering before it is visualized. This benchmark has been run under the X-Windows system as it needed to perform graphic I/O.
- **xanim x4.** Four concurrent executions of the previous xanim benchmark.

Table 1 summarizes the characteristics of each benchmark according to the parameters described in the first part of this subsection.

### 4.3 Performance Results

As this paper just tries to show that this mechanism is useful to increase the performance of large applications we will only present a selection of all the possible experiments. We will show the effect of the cache size given a cleaning threshold and the effect of this threshold given a cache size. Both parameters should be tuned for each system depending on the hardware and expected load. Anyway, even if the best configuration is not chosen, most reasonable configurations will imply a significant performance improvement.

#### General Performance Results

In this first experiment, we have configured the compressed cache using what we thought would be nice parameters. We have used a 1Mbyte cache and

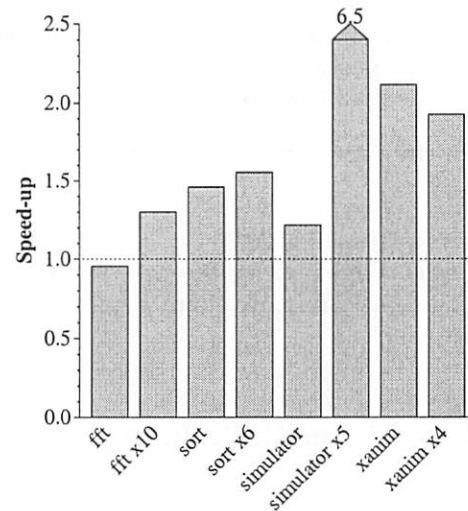


Figure 4: Effect of the compressed swap on several workloads.

a cleaning threshold of 50% of the cache. Using these values, we executed all the benchmarks and computed the speedup obtained when compared to the original swapping mechanism. Figure 4 presents these results.

In this graph we can see that all benchmarks but one observe a speedup between 1.2 and 2.1. This means that these applications run, at least, a 20% faster than with the original swapping mechanism and there are even executions where the applications half their execution time. The two exceptions to this rule are **fft** and **simulator x5**.

The first one (**fft**) achieves a speedup of 0.96, which means that it runs slower than with the original system. This slowdown is due to two basic factors. The first one is that the compression ratio is not very good and most pages cannot be compressed less than 2048 bytes. This means that it is quite difficult to place more than one page per buffer or disk block. The second reason is that taking memory from the application for our data structures and cache buffers has a significant effect on the application. Without this memory, the working set of some parts does not fit in memory anymore and the application pages much more than with the original system.

The second exception to a reasonable speedup is the execution of 5 concurrent simulations (**simulator x5**). This benchmark achieves a speedup of 6.5. Such an impressive improvement is due to its incredible compression ratio. As pages compress so well,

most swapped pages fit in the cache and nearly no disk access are needed.

These two exceptions will not be very frequent and we should expect a performance improvement between 20% to 100%, which is a significant gain.

Another unexpected result is the low speedup obtained by the `simulator` benchmark. As this benchmark compresses very well (6.7%), we expected to have a much more important speedup. The reason behind this behavior is the well behavior it has on the original system. As it swaps out many pages in very small periods of time, the original system can group them together before sending them to the disk and performs something similar to a batched write. For this reason, the gains we obtain by batching write operations together is also gained by the original system. This situation only happens in the original kernel when many pages are swapped out while writing the disk is busy. The kernel coalesces all these requests in a single one if contiguous. Anyway, this does not happen too often as we can see from the speedups obtained by the other benchmarks.

### Cache-size Influence

The second experiment tried to study the influence of the cache size in the performance of the new mechanism. To do this experiment we have run all the benchmarks varying the cache size between 256Kbytes to 4Mbytes and the cleaning threshold used in all these experiments was 50%. The obtained results are drawn in Figure 5. In this graph, we will not present the results of `simulator x5` as a curve with speedups of 6 would difficult the study of the graph.

The main observation is that there is nothing such as a perfect cache size for all benchmarks. It is clear that very large caches are no good as they take too many pages from the applications and they have to swap far too much.

Anyway, the important thing is that with reasonable cache sizes (around 1Mbyte) the performance of the applications is greatly improved due to the compressed cache.

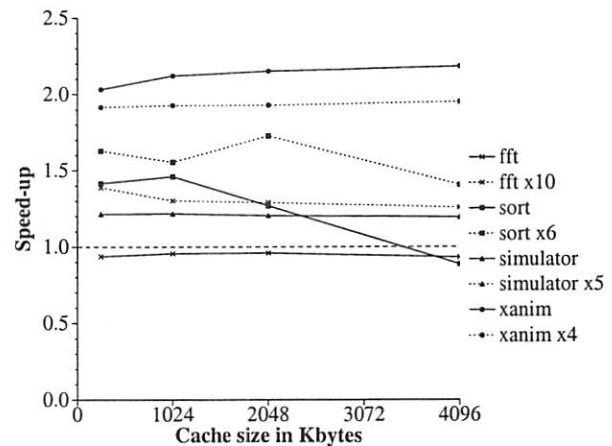


Figure 5: Influence of the cache size on the application performance.

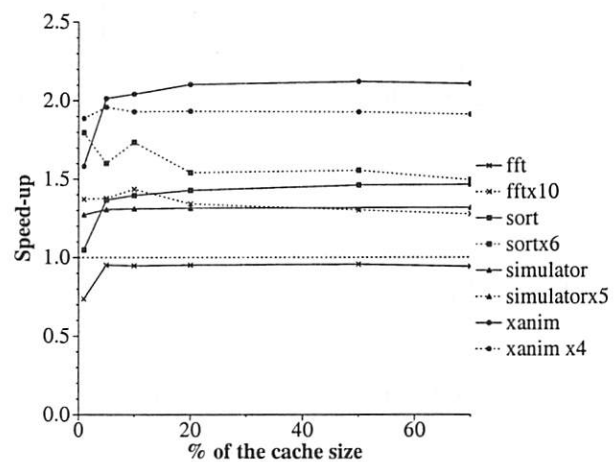


Figure 6: Influence of the cleaning threshold on the application performance.

### Cleaning Threshold

The final parameter is the cleaning threshold. This value defines the percentage of buffers that have to be *full* before a cleaning operation is started. To study the influence this parameter has, we have set the cache size to 256Mbytes and we have varied the threshold from 1% to 70%. The results obtained by all the benchmarks are presented in Figure 6. The performance of `simulator x5` is also not included in this graph for the same reason as in the previous subsection.

We can observe that this parameter cannot be too small. In this case, many small write operations are performed and the seek and search actions become an important overhead in these write operations.

When the threshold grows, the performance of the applications tends to increase as the disk latency becomes less important.

For very large values, the behavior of the system depends on whether the benchmark is a mono or multi-process one. In mono-process benchmarks, when the cache is being cleaned (synchronous cleaning), no other process accesses the disk and the benefits of large writes continue to be the a good issue. On the other hand, in multi-process benchmarks, while a process is cleaning the cache, another may want to swap in or out a page. If the cleaning operation is too long, the read/write operation has to wait for a long time and the performance of this process is also affected in a negative way.

The benchmark `sort x6` has a very unpredictable behavior. It does not follow any clear pattern. The reason behind this behavior is the large amount of I/O it performs. If it tries to write while the system is cleaning the cache, the performance is greatly affected. There is no way to avoid these collisions, but still the results are good enough.

It seems that the best value for this threshold is between 10% and 20%.

#### 4.4 Increase of the Swap Space

So far, we have only seen the performance benefits of compressing the swap area. As we mentioned in the introduction, this was the main objective of the project. Anyway, we also had a second objective that consisted on increasing the number of pages that could be placed in the swap area. A study of this objective is presented in this subsection.

Although the number of pages that fit in the swap partition depends on the compression ratio, it is not the only important factor. The fragmentation found inside the buffers will also be an important parameter in the final size of the swapping space. A system that leaves large unused portions in the buffers will not be able to place many more pages than the original system in the swap partition.

Figure 7 presents the size of the swap area that we would obtain if we could fill it with pages following the same compression ratio and the same fragmentation as the ones obtained in the benchmarks. In the figure, we can see that in most cases the system

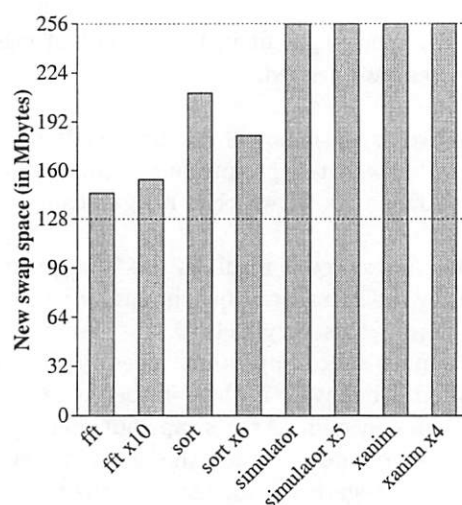


Figure 7: Size of the swap area that would be needed to have the same capacity as our 128Mbytes compressed swap.

increases the size of the swap partition more than a 50%.

We have limited the gain to 256 Mbytes as we have configured the size of the `virtual_swap_info` table to double the physical swap space. If a greater array were used, a larger swap space would have obtained with benchmarks such as `simulator` and `xanim`.

## 5 Related Work

Not much research has been done in the area of compressing the swap space. The *compression cache* proposed by Fred Douglass [4] is very similar, in essence, to our work, but some important differences can be found. In that work, the swap pages are also compressed and kept in a cache to increase both the size of the virtual memory and the performance of the applications that have to swap. One big difference between our work and the one done by Douglass is that the results we present are not so dependent on the compression ratio as they were. In the previous work, no performance gains were obtained with compression ratios worse than 30% while we obtain nice performance improvements with even a compression ratio of 62%. This might be either due to design issues or due to the improvements in the technology (compressing is much faster now). It is also important that the previous work lacked a study

on the kinds of read hits obtained in the cache. This study has led us to significant design modifications such as having two different paths: one for swapping in and one for swapping out. As we have shown in this paper, this distinction has obtained significant performance benefits. Finally, all their benchmarks were single process while we believe that multi-process benchmarks have also to be studied.

If we examine our work in a more general way we can divide it in two basic issues: increasing the size of the memory and reducing the average time needed to swap in/out a page. Let's discuss what has been done in both fields.

Following the idea of increasing the size of the memory, there are some commercial products that compress the physical memory. With these software mechanisms the applications believe that the system has a larger amount of physical memory. Anyway, the achievements obtained by such systems are not clear [8, 14]. The same idea has also been done in hardware with much better performance gains [6].

There have also been many proposals to decrease the number of disk accesses for swapping issues. For instance, some work has been devoted to minimize the number of pages that have to be swapped out. If the contents of a page is irrelevant to the application execution, this page does not need to be kept in the swap [15, 5]. In the same line, software has been developed to study the utilization of the pages and thus improve the programs and reduce the number of pages swapped in/out [12]. There has also been some work that tried to group pages when swapped out so that larger writes were done [1].

Finally, the approach of compressing information before sending it to the disk is widely used in database environments and in some file systems.

## 6 Conclusions

In this paper, we have presented a way to implement a compressed-swap mechanism that achieves significant improvement in the performance of lager applications. Most of them achieve speedups between 1.2 and 2.1 and there are some special cases where this speedup is even much higher.

We have also shown that, although the configuration affects the performance, it is not difficult to find a reasonable set of values that work well with all applications.

Finally, this mechanism has been installed in some Linux boxes in our department and the users are quite happy with this new feature.

## Acknowledgments

We would like to thank Professor J. Labarta and X. Martorell for all their help. Their interesting comments increased the quality of this work significantly. We are also grateful to E. Artiaga who helped us with some of the benchmarks used in while testing the proposed mechanism. Finally, we are also in debt to J. Fischer and E. Youngdale, which had the patience of guiding us into many important details of the SCSI driver for Linux.

## References

- [1] BLACK, D., CARTER, J., FEINBERG, G., MACDONALD, R., SCIVER, J. V., WANG, P., MANGALAT, S., AND SHEINBROOD, E. OSF/1 virtual memory improvements. In *Proceedings of the Mach Symposium* (November 1991), USENIX Association, pp. 87–103.
- [2] CARD, R., DUMAS, E., AND MÉVEL, F. *Programming Linux 2.0*. Editions Eyrolles, 1997.
- [3] CERVERA, R., AND CORTES, T. Swap primit: Disseny i implementació. Tech. Rep. UPC-DAC-1998-34, Universitat politècnica de Catalunya, Departament d'Arquitectura de Computadors, 1998.
- [4] DOUGLIS, F. The compression cache: Using on-line compression to extend physical memory. In *Proceedings of the Winter Technical Conference* (January 1993), USENIX Association, pp. 519–529.
- [5] HARTY, K., AND CHERITON, D. R. Application-controlled physical memory using external page-cache management. In *Proceedings of the V Architecture Support for Programming Languages and Operating System* (October 1992).

- [6] KJELSO, M., GOOCH, M., AND JONES, S. Design and performance of a main memory hardware data compressor. In *Proceedings of the 22nd Euromicro Conference* (September 1996), IEEE Computer Society Press, pp. 423-430.
- [7] KJELSO, M., GOOCH, M., AND JONES, S. Empirical study of memory-data: Characteristics and compressibility. In *Proceedings IEE, Comput. Digit. Tech.* (January 1998), vol. 145, pp. 63-67.
- [8] LEONARD, D. Magnaram 97 is no cure-all for the ram blues. C|net Special Report (<http://www.cnet.com>), October 1997.
- [9] LEVY, H., AND LIPMAN, P. Virtual memory management in the VAX/VMS operating system. *IEEE Computer* 15, 3 (March 1982), 35-41.
- [10] MOWRY, T. C., DEMKE, A. K., AND KRIEGER, O. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings of the 2nd International Symposium on Operating System Design and Implementation* (1996), USENIX Association.
- [11] OBERHUMER, M. F. Lzo v1.04. <http://wildsau.idv.uni-linz.ac.at/mfx/>, March 1998.
- [12] ROBINSON, E. M., AND LEIS, E. L. Page utilization in fortran and C programs. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications* (July 1998), CSREA Press, pp. I:206-210.
- [13] SMITH, A. J. Disc cache - miss ratio analysis and design considerations. *ACM transactions on Computer Systems* 3, 3 (August 1985), 161-203.
- [14] STEERS, K. Tune up your memory. *PC World* 24 (August 1997).
- [15] SUBRAMANIAN, I. Managing discardable pages with and external pager. In *Proceedings of the 2nd Usenix Mach Symposium* (November 1991).
- [16] VERGHESE, B. *Resource Management Issues for Shared-memory Multiprocessors*. PhD thesis, Sanford University, March 1998.
- [17] ZIV, J., AND LEMPEL, A. Compression of individual sequences via variable-rate coding. *Transactions on Information Theory* 24 (September 1978), 530-536.

# New Tricks for an Old Terminal Driver

Eric Fischer  
*The University of Chicago*  
enf@pobox.com

## Abstract

Users expect more out of command lines than they did fifteen years ago, but the terminal interface has not evolved to keep up with their expectations. With a few modifications, though, the terminal driver can provide every program with support for the arrow keys and several common Emacs commands.

## Introduction

After some significant improvements in 4BSD and System III in the early 1980s, the Unix terminal interface stopped evolving. Since then, even substantial rewrites like 4.4BSD and completely new implementations like Linux have given terminals the same old set of features without adding anything new.

Meanwhile, users have continued to demand better and better interfaces. With stagnation in the operating system, the responsibility for improvement has fallen to individual programs. The result is that some programs (the ones that do all the work themselves) have very good interfaces, while others (the ones that depend upon operating system services) have very bad ones.

It would be very difficult to make the kernel include all the editing features from Emacs or *vi*. These editors are large and complicated programs, and a reasonably complete imitation of either of them must also be large and complicated. Attempting to make the same kernel code work on different versions of Unix makes the complexity even worse.

But a complete clone of Emacs is more than most programs really need. As Kernighan and Plauger put it, "most users of a tool are willing to meet you halfway; if you do ninety percent of the job, they will be ecstatic." The standard kernel provides about fifty percent of what user testing shows that people want out of a command line editor. A few select enhancements will raise this figure to more than ninety nine percent.

Most importantly, it turns out, people want to be able to delete things: the previous character, the next character, the previous word, the whole line, and to the end of the line. They want to be able to move up and down a history list. They also want to be able to move left and right within the line they are editing, a character or a word at a time, and to the start or end of the line. Finally, they want to be able to complete partially-typed filenames and to clear the screen.

Other, more elaborate features are occasionally useful, but most users will never notice that anything is

missing if they can do the things listed above. Selecting this small set of features to implement also means that portability need not be a major concern, since the ideas can easily be applied to a different kernel implementation even if the code itself cannot be.

## Implementation

As you type characters, the terminal driver places them into a structure called (on BSD systems) the raw queue. This structure was not designed for elaborate editing, so there is normally no way to add characters to or remove characters from it other than at the end.

One way to work around this limitation without introducing an entirely different representation is to use two queues for the current input line. The raw queue still holds the characters to the left of the cursor, and a new queue, the editing queue, holds the ones to the right of it, if there are any. As the cursor is moved left, characters are removed from the raw queue and placed into the editing queue. When moving the other direction, the opposite happens.

So, for example, if someone had typed the line

```
sort file | un□q -c | sort -rn
```

and moved the cursor back over the *i* as shown, the raw queue would contain

```
sort file | un
```

and the contents of the editing queue would be

```
nr- tros | c- qi
```

The characters in the editing queue are in reverse order because it is really being used as a stack, not a queue. When the characters are moved back, one at a time, into the raw queue, they will again be in the correct order.

Since the characters after the cursor are kept in their own queue, inserting and deleting characters before the cursor can be done in exactly the same way as with the standard driver. As a result, most of the code never needs to know that the editing queue exists at all. Most of the parts that do know about it only use the two functions that move the cursor left and right.

The functions that move the cursor left and right update the structures in memory and the image on the terminal at the same time. The only control character that the updating routine uses is Backspace, so it should work with nearly any terminal.

## User interface

Once this infrastructure is in place, the next task is to make a user interface for it. Other programs have already established a de facto standard for what this interface should look like: people expect to be able to use the arrow keys and some of the control characters from Emacs. In practice, even diehard *vi* fans generally seem to be willing to put up with Emacs-style command line editing.

This is lucky, because a minimal version of Emacs is much easier to write than a minimal version of *vi*. A *vi* editing mode would be a good thing to add eventually, but I have given up on it for now because *vi*'s compound command structure and the awkward access to data in BSD queues made it too hard to do a good job.

Most basic Emacs commands, on the other hand, are very straightforward to implement. Control-B and Control-F, which move the cursor left and right, respectively, simply call the primitive function that performs this task. Control-A and Control-E, which move to the start and end of the line, do the same, but keep calling the function until it fails, which means that the end of the line has been reached.

The deletion commands are only slightly more complicated. Control-K, which deletes up to the end of the line, first checks to see how many characters there are after the cursor, moves forward that many characters, and then deletes backward the same number of times. Control-D, which deletes a single character, is cursed by being the same character that Unix uses for end-of-file. If there are any characters after the cursor, it deletes one; otherwise, it falls through to the normal input processing which interprets it as end-of-file.

These Emacs commands, incidentally, are only interpreted when the `L_EMACS` bit is set in the terminal control flags and the terminal is in canonical mode. This allows anyone who only wants to use the standard terminal facilities to disable the Emacs commands with `stty -emacs`. There is room for an `L_VI` flag for when a more ambitious version adds support for the *vi* command set.

## Compounds and arrows

Other Emacs commands and ANSI-standard arrow keys use multicharacter sequences beginning with ESC. So when the terminal driver is in Emacs mode and receives an ESC character, it sets a flag to record this

and then returns to wait for the next character to arrive. This is similar to the way the literal-next character is already handled.

If the next character that arrives is also ESC, it falls through into the normal processing, so a *cs*-style file-name completion facility can still be used by typing ESC ESC. If the character is `b` or `f`, then the sequence is the Emacs backward- or forward-word command and the cursor is moved backward or forward until a word boundary or the end of a line is reached.

If the character after ESC is `[` or `O`, then it is assumed to be part of the sequence for a VT100-style arrow key, and another bit is set to indicate this. When the following character arrives, if it is A, B, C, or D, then the appropriate arrow key action is taken.

It is a shame to hardwire these sequences into the program, but they are used by nearly every terminal, they are specified by an ANSI standard, and the code to support them is much less complicated than a more configurable version would be. The same features are still available on non-ANSI terminals by using equivalent Emacs commands rather than arrows.

## History

Since the function of the up and down arrows (as well as Control-P and Control-N) is to move through the history list, this is an appropriate point at which to introduce the history mechanism. In an earlier (1997) implementation, I put all the code to manage the history list directly in the terminal driver, but the experience convinced me that history is too complicated and takes too much memory to make it reasonable to put entirely inside the kernel.

In the current version, the real work of maintaining the history list is done by a daemon, *tytd*, which runs as a user process and makes *ioctl* calls to listen for instructions from the terminal driver. The driver can post requests for the previous or next item from a terminal's history list or to add a new line to the list. The daemon satisfies these by using additional *ioctls* to query and set the contents of a terminal's current input buffer.

Each process on each terminal has a separate history list so programs do not interfere with each other. Typed input is added to the history list only when the terminal is in canonical mode and only when echoing is turned on, so there should be no risk of passwords ending up in the history by mistake.

Like the Emacs features, the history list can be enabled or disabled for a particular terminal by setting or resetting the `L_HISTORY` bit using *stty*.

The new *ioctls* that were added to support the history features also turn out to have more general uses.

The header-editing feature in Berkeley *mail*, for instance, stuffs each header into the editing buffer by faking a series of keystrokes. When rewritten using one of the new *ioctls*, it is simpler, shorter, and less prone to mysterious bugs than the current version.

## Completion

Probably the most important element of an easy-to-use command line interface, and unfortunately the hardest to do well, is a completion feature that can automatically provide the rest of a partially-typed command or filename. There are several ways to implement this, none of which is completely satisfactory.

As mentioned above, *cs**h*-style completion still works with the modified terminal driver. The *cs**h* feature works by making the system return a partially typed line and then faking keystrokes to get the completed version back into the editing buffer. As the manual notes, this approach is “ugly and expensive,” and it requires each program that needs a completion feature to do all the work itself.

A variation on this theme gives control back to the user program by sending it a signal when the completion character is typed. The signal handler then uses one of the new *ioctls* mentioned above to retrieve the contents of the current input line. It completes the line and uses another *ioctl* to put the modified line back into the terminal driver’s queue. This approach still requires the cooperation of each program, and also requires the addition of a new signal to the system. The current versions of NetBSD, OpenBSD, FreeBSD, and Linux each have only one slot left for a new signal, and it doesn’t seem very nice to take the last one.

A third approach gives the responsibility for completion to the daemon that is already providing history services. This eliminates the need to include completion code in every program, but it also means that programs cannot tailor the completion routines to meet their specific needs. This is also difficult to implement on a BSD system, because there is no easy way for a BSD user program to find out the working directory of another process, and filenames cannot be completed without knowing this.

Because of these problems, I am still experimenting with other ways the operating system might be able to provide a generalized completion facility.

## Conveniences

Some keyboards have Backspace keys, some have Delete keys, some have both, and some have keys labelled Backspace that actually transmit the Delete character or vice versa. Most programs that do their own editing work around this by making both

Backspace or Delete erase the previous character no matter which has been set as the erase character. This is just as easy to do in the terminal driver as it is to do in any other program.

In addition, the modified terminal driver can automatically set the erase character appropriately whenever either Backspace or Delete is typed in canonical mode, so raw mode programs are also informed which key is correct. This special treatment for Backspace and Delete can be enabled or disabled by setting or resetting the `L_SETERASE` bit with *stty*. People who prefer to use Delete as their interrupt character will obviously choose to disable it.

Finally, as mentioned earlier, a surprisingly popular feature of *tcsh* is its ability to clear the screen when a user types Control-L. Since the terminal driver does not have access to the *termcap* or *terminfo* database, it does not know what control sequence (if any) will clear the screen. It does, however, usually know how many lines tall the screen is, and outputting that number of blank lines is not a bad substitute.

## Conclusions

I have been using various versions of the software described here since early 1997. During that time I have found it very useful to have command line editing features consistently available in every program. If you would like to try this software on your own computer, copies of the source code are available at

<http://pobox.com/~enf/ttyedit/>



# The Design of the Dents DNS Server

Todd Lewis

*MindSpring Enterprises*

*1430 West Peachtree Street NW*

*Atlanta, GA 30306*

*tlewis@mindspring.com*

## Abstract

Dents is a server implementation of the Internet's Domain Name System. Dents main features are a modular driver architecture, a CORBA-based control facility, a replaceable tree system, a clean design and good karma. Dents is free software, licensed under version 2 of the GPL. In this paper, I describe the design of Dents, concentrating on the innovations and evolutions it embodies, and including the future directions in which we hope to take the server. I describe some of the problems we've had. Finally, I summarize some lessons about server design which Dents reflects.

## 1. What is Dents?

Dents is a server for the Domain Name System, the system whereby information concerning host names, including, most importantly, their IP address, is communicated through the Internet. DNS is a hierarchical caching directory keyed by name, with an extensible set of attributes which are associated with names.

Dents is free software, released under the terms of the GNU project's General Public License, version 2. It is coded in ANSI C, or as close thereto as we can come, and is oriented towards POSIX-conformant and POSIX-like systems. Dents uses POSIX threads, and while it is possible to compile the server without threads, it is not recommended, since several major features do not work without threads. Dents has a modular driver architecture, which permits various means to be used to look up names, and it includes a CORBA-based control facility, which allows administrators to control a running server. Dents should work on any modern unix-like system which supports shared libraries and threading; a win32 port is not out of the realm of the possible, although such is not presently planned. Primary development happens under the Linux operating system.

Dents as a project was started by Todd Lewis in early 1997. Johannes Erdfelt wrote the majority of the code,

and following our first public release in late 1998, Greg Rumble joined the team and has contributed significantly since then.

Dents was inspired by some particularly unpleasant experiences of the author in dealing with the DNS systems built at MindSpring. Thanks to its virtual web hosting product, MindSpring was then and is now one of the largest registrars of domain names in the world, and dealing with serving a large number of zones has been and remains a trying problem.<sup>(\*)</sup> Dents grew from frustration with how unfriendly existing DNS technology was to people trying to build higher-level management systems on top of the basic server. Better would be a server which you could reconfigure in a serious way without restarting, one which would allow the use of relational databases and other external systems to implement the underlying data management and retrieval functions, and the ability to update zones quickly and with a minimum of effort. Out of these desires grew some evolutionarily new ideas for how servers should be built. This paper describes some of these ideas, our experiences implementing them in Dents, and what we have learned from the experience.

## 2. The Driver Mechanism

One major feature of Dents is our modular driver mechanism. Conventional name servers simply read in all zone data at startup and store it in core memory. In cases where one has a large number of rarely-read records, this is a very sub-optimal use of resources. We wanted to be able to use a relational database as the underlying storage mechanism for DNS data, translating DNS queries into SQL queries and the reverse for answers. Further, we wanted for this system to be flexible enough that we could substitute any underlying engine for answering DNS queries, and clean enough that these engines would remain useful through multiple revisions of Dents itself.

The conceptual model which we settled into was similar to the notion of file system drivers within unix-like

operating systems. We decided that we wanted to "mount" zones at certain mount points within the hierarchical space, that each underlying zone would have a type and that it would be handled by a driver corresponding to that type. The driver serves to hide the actual details of the implementation and adapt them to a common interface, so that all instances have the same behaviour, regardless of the underlying technology.

It comes as no surprise, then, that Dents driver modules look fairly similar to file system drivers. There is a finite and well-defined set of functions which a driver must support. When a driver is loaded, these functions are used to populate a structure full of function pointers; this common interface allows all zones to behave identically, just as all file systems behave identically. Certain features are optional, and so if a driver does not support them, then the attempted use of them simply returns an error. (Symbolic links in the case of file systems, resource record addition and deletion in the case of Dents drivers.)

Dents uses the unix *dlopen()/dlclose()* mechanism for loading drivers. These drivers are identical to shared libraries, but instead of being linked in by the loader at start time, they are loaded by Dents when zones are added; they are demand loaded, so no explicit load is necessary, and they are reference counted and automatically unloaded when all zones using them are removed. The actual code was modeled after the GTK widget set's theming mechanism. To aid in robustness, one can specify at compile time for certain modules to be compiled statically into the server; the only difference internally is that statically-included drivers are never unloaded.

All zones in Dents are served using this mechanism; we rewrote both of our system components which serve zone data into modules. They are:

**mod\_stdtdb:** RFC-1035, section 5, specifies the format for transferring zones between name servers. This format has historically served as the native storage medium for zone data with other name servers. This module makes Dents behave just as traditional name servers do, reading in zone files in this format and storing them in an in-memory structure, looking up records in this structure later to answer queries. Although this is not our preferred method for running servers, it is a popular choice, and so we feel compelled to offer it, at least for migration purposes.

**mod\_recursive:** The DNS system relies heavily on local servers to perform DNS queries on a proxy basis

for local clients, in turn caching the results and using that cache for future requests for the same name. This module fills that role, allowing Dents to serve as a recursive name server for the root (i.e., the entire space with the exception of any local authoritative zones) or for any other zone. Answers are stored in a tree, which is then periodically purged. We hope to turn *mod\_recursive* into a very well-tuned cache; historically, DNS caches have simply never purged data, the system failing when system memory is exhausted. We aim to do better.

Further, we have several other modules in development by the Dents team:

**mod\_frl:** One particular case where the behaviour of traditional name servers is clearly suboptimal is in the case of *in-addr.arpa* zones. Internet service providers have large banks of modems, each of which usually has an IP address associated with it; i.e., customers who dial into that modem and negotiate a PPP session will receive that IP address. These IP addresses must be associated with names through the *in-addr.arpa* zone, or else certain classes of internet services will not work; e.g., certain FTP servers will not allow users access unless their IP has both forward and reverse name resolution.

At MindSpring, none of us in the Engineering department volunteered to name our several tens of thousands of modems individually, and so we came up with another solution: we name them algorithmically. If you look up the IP address *247.2.192.209.IN-ADDR.ARPA*, you will get back the name *user-38s00nn.dialup.mindspring.com*. This is simply the base-36 encoding of the IP address; the reverse of this query, i.e., looking up this name and getting back the IP address, works as well.

This is all very nice, but with a conventional name server we must generate several tens of thousands of these name/IP pairs and load them into the fairly expensive error-correcting system memory on our name servers. This seems rather silly for names which are generated algorithmically.

*mod\_frl* performs this algorithmic translation on the fly, allowing one to serve a very large number of zones with a comparatively small amount of memory. (The memory size of the module is constant; it doesn't even pay attention when you tell it that it is responsible for a new zone, simply answering any and all queries the server sends its way.) The code to perform this is 86 lines long, and the binary module takes a little over 4k

of memory. We think that this is a particularly good example of how a modular approach to serving DNS data can result in a significant advantage in metrics which are important to administrators.

**mod\_bdb:** Another fairly simple module is the Berkeley DB module, presently under development. The Berkeley database is a very simple and well-tested associative database, offering key=>value associations using either hash tables or btrees. We simply prepend the query type and a period to the record name to gain our key; for requests for all records for a given name (query type "\*"), we store a record which contains pointers to all other records for that name. This is a good example of how a flexible mechanism allows you easily to leverage the efforts of others; Berkeley DB is very easy to program, and we get a very fast mechanism which is well tested and which even supports transactional updates. (More on transactional updates below.)

**mod\_covert:** Cheswick and Bellovin suggest, in their book "Firewalls and Internet Security", that one could use DNS as a covert channel for tunneling data through supposedly secure environments. It would be an interesting exercise to write a module to accomplish this goal. Since many legacy DNS servers do not expire entries from their cache, this approach would have the unfortunate effect of crashing many of these servers, and so it would have to be used with care.

Finally, several outsiders are developing modules for Dents:

**mysql:** several people are working on implementing a module which uses mysql, a relational database management system, to store DNS data. When the module is passed a query, it communicates with the RDBMS via SQL to discover enough information to answer the query. It then formulates the answer.

The exciting thing about modules which use databases as their functional substrate is that it takes a class of tasks at which the DNS system is not very good and at which databases are good, and it transfers responsibility for these tasks from the DNS system to the database system. Specifically, large portions of the DNS standard deal with how zones are replicated across multiple servers. This replication happens wholesale and is unauthenticated. Major effort has been put into loading (I

would call it overloading) the DNS system to allow incremental zone updates and for updates to be secure. Updates are still not transactional, and there are at present no plans to make them so. (More on transactional updates below.) There are a number of commercial database systems which have long since solved these problems to a degree of completeness and reliability that DNS will never approach. This is yet another reason I believe that most zone data is better off being stored in a database rather than in the way zone data is conventionally stored.

mysql is popular in certain circles, but I would very much like to see other modules in this vein written: one for PostgreSQL, which is the preeminent free relational database, and some for the various commercial relational databases now available under Linux and other unices, including Oracle, DB2, Informix, Sybase, and Interbase.

**dhcp integration:** One very popular feature of Microsoft's DNS server for Windows NT is its integration with the Microsoft DHCP server. One developer is working on a driver module to integrate Dents with the ISC DHCP server. This module will take updates that happen in the DHCP server when a new DHCP lease is issued or an old one expires, and populate the DNS space with corresponding information on the machines dropping into or out of view.

(As an aside, Johannes Erdfelt is convinced that Dents can serve as a fairly generic server for associative data, serving such protocols as DHCP and LDAP in addition to DNS. This is why we segregate DNS-specific code from other server code in the code base. This is so remote a prospect at this point that I will mention it no further.)

**other modules:** One user is investigating using Dents to export a database of HAM radio operators via DNS. Another set of users is looking to use Dents as part of a project to adapt their DNS setup to use their own underlying database to deal with bandwidth-constrained links between their DNS servers. Others have expressed interest in Dents as part of various dynamic-IP naming projects. Our goal was to produce a very flexible system amenable to whatever tortuous uses people wish to apply the Domain Name System, and we seem to have succeeded with this.

### 3. The Tree System

After the driver module system, a second major architectural piece of Dents is our tree system. This component actually snuck up on us; we did not plan for it to be a major part of the Dents story.

Initially, Dents used a home-grown implementation of red-black trees to store its hierarchical data. (Which data is a lot for a hierarchical directory service.) We also used it for several internal data structures. The initial implementation dated from very early in the project, and we had several problems with it: for one, it was buggy, and for another, the caller managed all of the memory for record structures, which was a big mistake. Greg Rumble's big contribution to the project has been in working on this issue.

We realized that our initial code had problems and, rather than simply rewrite it, we decided to segregate it from the remainder of the server and make the red-black tree engine replaceable. We decided also to attempt this separation for several reasons.

First, we wanted completely to isolate the internals of the tree system from the rest of the code, to enforce good layering between server code and the tree code. Isolating the routines would allow us, e.g., to *mprotect()* internal memory upon leaving the tree code and *munprotect()* it when reentering the tree code, allowing very easy determination if any external code was touching the internals of our tree system. Good coding practice would accomplish this goal, but good programming practice can sometimes be in short supply, and mandatory discipline makes a fair substitute.

Second was the issue of memory usage. Unlike our major competitor, we had no problems growing past 64k zones as of version 0.0.1 and subsequently. Further, our implementation is plenty fast in the rough tests we have run on it. However, it consumes a lot of memory, almost twice as much as our competitor for an identical configuration. While this might be acceptable for certain installations, some others may be more memory sensitive. Rather than forcing one decision down our users' throats, we thought that having a plug-gable tree system would allow them to choose the tree engine which best suits their needs.

Third and most important was the issue of performance. The only real performance barrier in our main competitor is its use of a single hash table to store all DNS information. We believe that this is a design mistake and a competitive vulnerability. If you were to construct a histogram of DNS names and their hit counts, you would see a very tall spike at one end, representing

a few names which receive a very, very large number of queries, rapidly flattening out to a very long, flat tail which represents the remainder of the DNS space, records which are queried infrequently if ever. Using a hash table to store data of this profile guarantees that you will not gain the benefit of modern computer hardware architectures, with their large caches connected to (relatively) slow main memory. One thing which we hope to accomplish once we can plug in new tree engines is to implement a heavily cached engine, which will use layered caches to utilize the underlying hardware in the best possible way, allowing frequently-asked-about names to sit very high in the machine's cache hierarchy. We believe that the speed gain of this move will in the aggregate more than offset the additional cost of maintaining the cache, which can be accomplished mostly during idle time on the server. This comprises the core of our strategy for being the fastest name server.

Finally, almost as an afterthought, driver modules often have a need for the sort of functionality which our tree engine exports. As a courtesy, we would like to export this functionality to them, both to maximize code reuse, but more importantly also to minimize instruction cache invalidations as the server jumps from server code to driver module code and back to server code.

As I write this, we have just released version 0.0.3 of Dents, which embodies our preparation for performing this modification to the server. We are now embarking in earnest on implementing this code separation in the server. It will be reflected in our next release, and at that point work will begin on building alternative tree engines.

#### 4. The Control Facility

One feature which we hoped to have from the first day of the project was an administrative interface whereby we could interact with a running server, query it in detail concerning its state, and instruct it to perform various actions. Specifically, we wanted to depart from the stale convention of using configuration files read at server start as the exclusive means of determining the server's configuration. This convention causes downtime for the sake of configuration changes that do not themselves require downtime, and when your servers are very large, i.e., they serve a large number of zones, the cost of restarting the server is far from negligible. Even with small servers, such unnecessary outages offend the sysadmin aesthetic; this is another case where Dents reflects, we hope, the sensibilities of a sysadmin instead of those of a developer. We do not believe that

restarting the server should be required at all to make changes to the server's configuration, and we designed Dents with the goal of making this the case. For this reason, we resolved to include in Dents a control facility to serve this purpose.

#### 4.1. Inspiration

This feature is far from an original one; many other systems have allowed administrators to query and control them while they are running. Of course, the all time champions in this are commercial relational database management systems; these usually let you tweak any knob which is not inherently untweakable in a running system. The database community may have certain failings, but their attitude towards downtime is sufficiently paranoid, and is therefore laudable.

Somewhat closer to the traditional backyard of Usenix, two particular systems stand out as having inspired our approach in Dents. The first is the XNTP suite from the University of Delaware. Their xntpd facility does a very nice job of allowing an administrator to configure a running xntpd daemon. However, it does this by (ab)using the NTP protocol. We did not want to use the DNS protocol to control Dents for several reasons, which I discuss below.

The second inspiration for the Dents control facility was the kadmin interface of MIT's Kerberos distribution. kadmin is really the closest thing to our control facility of anything out there. kadmin uses ONC-RPC to talk to the server, rather than overloading the Kerberos protocol. Because of this, it gains several advantages. First, it benefits from the IPC infrastructure, and the IPC infrastructure benefits from it. Partially thanks to kadmin, Kerberos-protected ONC-RPC exists and is available for others to use; if this work had been done in a kadmin-specific way, then no one else would have benefited from it. Second, because kadmin uses ONC-RPC, they do not have to worry about handling the underlying wire protocol; all they see is an API, with the RPC package taking care of the details. Third, an RPC environment allows for much more expressive administrative interfaces. Hand-rolled protocols, because of the amount of work involved, inherently limit the expressiveness which they allow; the more expressive you are, the more work you make for yourself to handle the protocol in your code. RPC does not really suffer from this; the programming cost you see is the cost inherent in your interface design and nothing more. These considerations weighed heavily when we designed the Dents control facility.

#### 4.2. Why administrative protocols should be divorced from regular protocols

Before talking further about the Dents control facility, I would like to explain why I think that shoehorning these interfaces onto regular protocols is usually a mistake. First, overloading the regular protocol can damage the protocol. DNS has the interesting property that the addition of new resource record types is dangerous; because of the scheme used for compressing data in DNS packets, new RR types can render packets including that RR type undecipherable to older agents which do not understand that type. Even worse, if support for server control is included when the protocol is designed rather than added as an afterthought, (the later usually being the case), then the protocol is almost necessarily made more complex, introducing cost on all users of the protocol for the benefit of a few. Such additions can even be for the benefit of none, if the needs of the administrative community outstrip the ability of the protocols in question to handle them. Administrative needs can change significantly over time; one need merely compare sendmail to qmail to understand the variations possible in configuration needs among implementations of the same protocol. However, the protocols themselves should be timeless, or at least very slow moving, so as to get the maximum benefit of the programming effort expended in implementing them.

Second, regular protocols should not be overloaded to transform them into control protocols because they often make really bad control protocols. A prime example of this is updates in the Domain Name System. DNS is a stateless protocol, and as such it is virtually impossible, without butchering the protocol, to introduce transactional semantics. However, certain groups of domain updates (such as removing one machine from a DNS rotor and introducing a new machine to that rotor) really need to happen atomically. There is no middle ground here: either you perform all sorts of gross changes to the fundamental nature of the protocol, or you settle for an inferior administrative interface. Finally, the security semantics of the functional and the administrative interfaces are usually radically different from each other; services offered publicly or semi-publicly usually need very little if any authentication, whereas administrative interfaces usually need very stringent authentication and often privacy. This is another case where an impedance mismatch between the two divergent needs means that no system can truly satisfy both.

### 4.3. Implementation

Initially, we implemented the Dents control facility using a line-oriented protocol similar to POP or NNTP. However, we soon decided to switch to some flavor of RPC for the facility. We examined ONC-RPC, DCE-RPC and CORBA, finally settling on CORBA.

Our decision to embrace CORBA for this function had several reasons behind it. Most of the reasons apply to any form of RPC.

- The details of the protocol are hidden from you, along with all of the work required to handle them.
- When one considers the human factors involved, from inaccurate documentation to imperfect implementations, they are much less error-prone than hand-rolled protocols.
- They come with lots of free add-on services, such as security and transactions, that are difficult and usually unpleasant to implement yourself for a single project; because this effort is shared across multiple projects, the resulting code is usually much better.
- Finally, RPCs are much more expressive than hand-rolled protocols, precisely because you are shielded from the programming which has to occur to support a complex client-server interaction.

We finally chose CORBA over the other forms of RPC because:

- it supported exceptions, rather than the more restrictive, conventional C style of signaling error conditions;
- its associated services were much broader, offering more value to us than ONC RPC, which sometimes has security associated with it and nothing else;
- there are free CORBA implementations which are being very actively developed, unlike wither ONC or DCE RPC;
- finally, CORBA itself is undergoing active development, from the core protocol to the CORBA services, whereas DCE RPC is, so far as I can tell, dead, and ONC RPC is developing very slowly.

With our recent public release of version 0.0.3 of Dents, we included this new CORBA-based control

facility. During the upcoming release cycle, we hope dramatically to increase the range of functionality available through the control facility, now that the infrastructure is in place and working.

### 4.4. What a control facility makes possible

The control facility made an initial project feasible: SNMP support. RFCs 1611 and 1612 specify the DNS server and client MIBs. For the past few years, no DNS server to my knowledge has supported these MIBs. Because of the control facility, adding support for these MIBs to the University of California, Davis SNMP server was a very easy task. I simply added a new module to the server, as the UCD docs describe how to do, and had that module make a connection to Dents through the control facility. When requests for statistics come in, the SNMP server simply makes control facility calls and returns the results to the SNMP clients. I am an inexperienced C programmer, I had this scheme working and answering queries in an afternoon; this is a good indication of how powerful CORBA is as an easy way for doing IPC. This compares favorably in many respects with alternate schemes for IPC between SNMP servers and other system servers.

The second goal we have for the control facility is that users (i.e., administrators) be able to control their servers through automated scripts. A good example of this is, again, MindSpring's web hosting environment. When web hosting accounts are set up, often times there is no human intervention. Customers sign up on a web page; the billing system validates their credit card; the billing system then informs the web hosting system to create an account; the web hosting system creates both the WWW part and the DNS part of the account, and then informs the user via email that his account is ready to be used. It would be nice if these scripts could contact a running server and tell it to add a new zone, immediately, without having to restart it. The time to restart the DNS servers is, right now, the single biggest source of delay in creating web hosting accounts at MindSpring, by far. Similarly, when customers do not pay or exceed their bandwidth limits, it should be possible to turn them off immediately, rather than having to wait for the next scheduled server restart. These operations should be strongly authenticated.

The final goal for the control facility is to enable graphical administrative clients. Ironically, automated interfaces and graphical interfaces have the same requirements, and textfile-based interfaces are hostile to both. (For some reason, many people think that

textfile-interfaces are friendly to automated environments; of course, nothing could be farther from the truth.) It should be possible through a graphical client, using the control facility, to set up a new server, to add or delete zones, to populate those zones, to examine the change logs for a zone, to set up primary and secondary servers, and all of the other things that admins need to do to

## 5. Future Directions

**Transactional updates:** One particular area where the approach of using an out-of-band control facility really shines is in the area of updates. Many users have the need for several related updates to DNS to be batched together and processed atomically; they need transactional semantics for updates. The DNS protocol itself, being stateless, is singularly unsuited for providing transactional semantics. We plan on offering users atomic updates to DNS data using the control facility and CORBA Transaction Service; this is another case, in addition to security, where having a rich suite of services available within the context of CORBA really pays off.

**More Drivers:** We will be working hard during the coming months to assist people other than the core developers in writing modules. We hope to use the driver system to allow developers to develop systems to meet their own needs with a minimum of development effort. We have talked about embedding interpreted languages as modules, so that less experienced programmers can write scripts to formulate answers instead of having to program in a compiled language which is compatible with our binary interface.

**Tree Code:** The separation of our tree code from the rest of the server and the ability to plug replacement tree engines into the server will allow a good deal of experimentation with different performance profiles for the server. Specifically, it would be good to have both a low-memory engine and a high-performance engine available for users to select as their needs dictate.

## 5. Problems we've encountered

We use the GNU project's automation tools for managing our build environment, specifically automake, autoconf, and libtool. We have decided to use the latest versions of these tools; this has caused a good deal of difficulty for non-experts trying to compile the system. As project lead, I feel torn between the developers who say that the tools really help them and the users who say that the tools really hurt them. As we get closer to

a final rollout of v1.0 of the code, we will lean more towards helping users and away from favoring the developers.

We encountered a problem where certain of our system structs, which were included in modules, had certain portions `#ifdef`d out if the server was not compiled with POSIX threads enabled. If the modules and the server were not compiled with the same setting for POSIX threads, then they were binarily incompatible. We solved this problem by declaring that POSIX threads need to be enabled as the standard, and that non-POSIX-thread-enabled builds were only for debugging and were aberrant.

Finally, we have simply taken too long to get Dents out the door. Sometimes we had good reasons and sometimes we didn't. Once we did the public release, then things really started moving with the development effort. We should have made a public release much sooner.

## 5. Lessons Learned

### *Divorce functional from administrative interfaces*

Mixing functional and administrative interfaces damages your functional interface and cripples your administrative interface. Server designers should have the courage to divorce the two, and use the resulting freedom to deliver full-featured interfaces which allow admins control over their servers. Kerberos and relational databases are a good model to follow here.

### *Use CORBA*

The difference between rolling your own protocol and using CORBA is like the difference between writing programs in assembly and writing them in a higher-level language. Just like the later case, surprises lurk when it comes to performance. Just as compiled programs are often more efficient than hand-programmed assembly, CORBA, by virtue of its pass-by-reference nature, can often achieve network efficiencies that hand-rolled network protocols can not. The goal the designer is trying to accomplish is usually something very close to a remote procedure call anyway and can be adapted to the RPC paradigm without problem. CORBA makes many great things possible.

### *Servers as machines*

Here I speak speculatively and very explicitly for myself and not for my fellow Dents developers, some of

whom might not agree with me on this issue. Increasingly, I come to the conclusion that a good model for servers is that they be machines, in the formal sense. Servers should not be responsible for configuring themselves, just as operating systems are traditionally not responsible for configuring themselves. Rather, like operating systems, servers should simply export functionality and bootstrap themselves only as much as it takes to allow outsiders to utilize their functionality, and they should allow their state to be read completely. In this way, the start and stop of the system are simply temporal holes in the provision of service, not configuration points. Under this rule, downtime need not be unnecessarily spent on configuration changes. Today, functional requirements about configuration needs, which should properly be embedded in the source code as assertions regarding the state of the system's configuration, are instead "enforced" by opportunistic order of configuration routines in the bootstrap section of the server. This is a horrible area of unfixed and unfixable bugs, as often these requirements are never documented. These requirements should be properly reflected in the code, ala: "ERROR: can not load new zone until system module path set; can't find modules!"

The conservatism of the unix community in the design of system servers has led to outright stagnation; new ideas are very far between these days in how servers are designed, and this is a real shame. Our real hope with Dents is not so much to take over the DNS space as it is to introduce and promote some new ideas about how servers should be written. It is our hope that they will be successful for us, and that our success will inspire others, not only to emulate our techniques, but to emulate our innovation.

## Acknowledgements

Thanks, of course, to Johannes Erdfelt and to Greg Rumble for doing all of the hard work on Dents. Throughout the life of this project, MindSpring Enterprises has employed each of the three major authors of this project. While MindSpring has never officially supported the project, they have made project resources available to us and been very understanding of those few times when Dents has come first and work second. Additionally, our coworkers at MindSpring have been a great, unlauded help to us as we struggle to learn how to build servers. I am grateful for their support. Finally, I'd like to thank Paul Vixie and all of the other contributors to the DNS standardization process for creating such a fun protocol to work on; the entire

Internet owes them a debt for such a functional name system.

# THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

## SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

### Member Benefits:

- Free subscription to *login*, the Association's magazine, published eight-ten times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and C++, book and software reviews, summaries of sessions at USENIX conferences, and Snitch Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, via the USENIX Online Library on the World Wide Web.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as object-oriented technologies, security, operating systems, electronic commerce, and NT – as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Discount on BSDI, Inc. products.
- Discount on all publications and software from Prime Time Freeware.
- Savings (10-20%) on selected titles from Academic Press, Morgan Kaufmann, New Riders/Cisco Press/MTP, O'Reilly & Associates, OnWord Press, The Open Group, Sage Science Press, and Wiley Computer Publishing.
- Special subscription rate for *The Linux Journal*, *The Perl Journal*, *IEEE Concurrency*, and all Sage Science Press journals.

### Supporting Members of the USENIX Association:

C++ Users Journal	Internet Security Systems, Inc.	Questra Consulting
Cirrus Technologies	Microsoft Research	Sendmail, Inc.
Cisco Systems, Inc.	MKS, Inc.	Server/Workstation Expert
CyberSource Corporation	Motorola Australia Software Centre	TeamQuest Corporation
Deer Run Associates	NeoSoft, Inc.	UUNET Technologies, Inc.
Greenberg News Networks/MedCast Networks	New Riders Press	Windows NT Systems Magazine
Hewlett-Packard India	Nimrod AS	WITSEC, Inc.
Software Operations	O'Reilly & Associates Inc.	
	Performance Computing	

### Sage Supporting Members:

Atlantic Systems Group	Mentor Graphics Corp.	SysAdmin Magazine
Collective Technologies	Microsoft Research	Taos Mountain
D. E. Shaw & Co.	MindSource Software Engineers	TransQuest Technologies, Inc.
Deer Run Associates	Motorola Australia Software Centre	Unix Guru Universe (UGU)
Electric Lightware	New Riders Press	
ESM Services, Inc.	O'Reilly & Associates Inc.	
GNAC, Inc.	Remedy Corporation	

For further information about membership, conferences or publications, contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA.

Phone: 510-528-8649. Fax: 510-548-5738.

Email: [office@usenix.org](mailto:office@usenix.org).

URL: <http://www.usenix.org>.

